A Reliable Communication System for Xilinx MicroBlaze FPGA Systems

User's Guide

Patrick E. Akl
American University of Beirut
Computer and Communications Engineering

Prepared at the
University of Toronto

June – July 2004

Table of Contents

Acknowledgments

I would like to thank all the people who helped me take part in this project. Special thanks to Professor Paul Chow for giving me that chance, and to Professor Mazen Saghir, for his feedback and academic advising. I would also like to thank all the students involved in the Computer Research Group at the University of Toronto for their warm welcome.

Project Information

1. Project Definition

Reconfigurable computing systems are becoming popular as an alternative way to solve many problems. The main advantage of using them is the fact that they provide an efficient computing platform in terms of performance, and that they save us from VLSI development, which is a risky, costly and very long process. In fact, microprocessor speeds often improve faster than it is possible to build an application specific processor to solve the same problem, which means that by the time the processor is built and ready to be tested, the software implementation using a current microprocessor can run at about the same speed.

With Field Programmable Gate Arrays (FPGAs), it is possible to implement systems and test them quickly relative to the long process of VLSI development. Another advantage is also the cost of implementation which is clearly reduced. Moreover, the design is generally written in hardware configuration code such as Verilog or VHDL and can be used on many other FPGAs. Programming using high level programming languages is also possible, and the compilation process includes various optimizations, some of which can target space or speed constraints.

The goal of the project is to use FPGAs to build a multiprocessor like machine. The Application being developed to test the system simulates biomolecular interactions and performs many computations on various forces, energy levels….These types of applications usually run on large systems like clusters or supercomputers. The final system can be called an Application Specific Reconfigurable Multiprocessor, which means that the system is built with specific knowledge about the problem. For instance, to compute various forces and other parameters, custom hardware is being developed in order to speed up the computation.

This report described the implementation of two versions of a general purpose reliable communication protocol that will be used to interconnect the FPGA Boards. The communication system runs over Ethernet 802.3, and the number of nodes in the system can be configured expanded using switches and hubs. An additional implementation of a sniffer node was done for simple network monitoring.

2. Hardware and Software Used

The Hardware components used in this project are the following:

- *Xilinx MicroBlaze and Multimedia Boards*: The MicroBlaze and Multimedia Board, developed by Xilinx corporation, is designed to be used as a platform targeting multimedia applications such as audio, image and video processing, networking…, it contains a Virtex II Pro FPGA as well as 5 external memories of 2 MB capacity each. The board supports PAL and NTSC television input and output, true color SVGA output, and an audio coder / decoder with power amplifiers. For our project we will be mainly using Ethernet for inter-FPGA

communication, and RS 232 serial interface for debugging and verification of the system. In addition to that, several DIP switches and output leds were used in order to control the operation of the system and for debugging.

- *Networking Components*: We mainly used a 10 Mbps Ethernet Hub and a 100 Mbps Ethernet Switch as well as various RJ45 cables to test our system.

- *Debugging and Monitoring Components:* In order to debug our system and test the consistency of data and proper operation of various characteristics of the communication protocol, a serial port on the board was used as an output and fed to the input of a computer on which a HyperTerminal window was configured to read and display the data fed by the board.

- *Sniffer Node for monitoring:* A simple system was implemented on one board for use as a sniffer. That is it is connected to a hub and reads all frames on the network without the network being affected by it. It can be configured in two modes to determine the monitoring level (heavy or light) using a DIP switch.

  The Software components and programming languages used in this project are the following:

- *VERILOG and VHDL:* Hardware Description languages that were used in some user defined cores used in implementation project.

- Xilinx ISE 6 as well as Xilinx Platform Studio 6,2i*:* these are the software tools used to compile our programs, configure the hardware that will be placed in the FPGA and perform various synthesis steps to obtain the bitstreams that were downloaded on the FPGA. The programming language used for device driver programming is a Xilinx version of the C programming language. It mainly contains some new definitions of data types to make driver configuration user friendly.

3. Design Constraints

Before starting the design of the system, we made some decisions about the main constraints we need to try to satisfy in our final system, these are the following:

- *Node Identification*: An Advantage provided by the multimedia board is that the MAC Address of the EMAC module can be configured by the user using high level functions. In reality, the EMAC module contains a hard coded MAC Address to satisfy the IEEE requirements. We were not, however, dealing with this MAC address. The user has the capability to select, via high level driver functions, the local MAC Address of the board where the software will be downloaded.

- *System Scalability*: the system is able to accommodate any number of nodes and expanding it using hub and / or switches only require a minor straightforward modification in the code definitions contained in one of the headers of the system.

- *Message and Frame Parameters*: The biomolecular computation program used as a test application for the FPGA based multiprocessor system would

only require transfer of messages of sizes of few tens of bytes (mainly transferring space coordinates, force and energy values, requests for computations…), hence without messages of size greater than 1500 bytes which is the maximum allowed by the Ethernet Protocol, we avoid the problem of fragmentation and reassembly. However, we decided to design a general purpose looking system for reusability for any type of application and any message sizes. Hence a higher layer header was needed to make the fragmentation and reassembly possible.

- *Data Consistency*: the Ethernet Protocol provides a 32 bit CRC known to mathematically detect almost all of the errors that might occur in the transfer of frames, hence we did not need to design any other "check" sequence. Device is built such that only valid frames with no bit errors values are passed to the user.

- *Reliability of Communication*: Any system needs reliability in order to function properly. In networking, for several reasons, frames can be corrupted, and even totally lost. So there has to be a way to detect these events from the sender's side and have a scheme for retransmission of frames to make sure the receiver got the message correctly, and the sender be notified of the success or failure of the operation. Moreover, the receiver has to have a way of distinguishing new frames from old ones, i.e. not accept a same frame twice.

- *User interface*: all the user should see from the system is a general "send function" as well as an interrupt driven function where he can fetch the frame payload and have some variables to demultiplex it properly.

- *Performance*: The protocol would probably be also used for file and real time video transfers between FPGAs. So various implementations' effect on the streaming bitrate was considered in the design decisions.

- *Dual MicroBlaze System*: the Dual MicroBlaze system consists of two MicroBlazes: a "user" MicroBlaze and a "network" MicroBlaze. The MicroBlaze dealing directly with the network is connected to the EMAC controller using the OPB, and the user MicroBlaze is "free", i.e. having an OPB with no modules connected to it except for the interrupt controller and it is only connected to the "network" MicroBlaze with FSL (Fast Simplex Links) and via a signal that acts as an interrupt signal from the "network" MicroBlaze to the interrupt controller of the "user" MicroBlaze.

  Note: two systems were implemented, one consisting of a single MicroBlaze, and the other one consisting of a dual MicroBlaze system, with MicroBlazes communicating with each other using fast serial links (FSLs).

Preparation Experiments

In order to familiarize ourselves with the tools we will be using as well as the VIRTEX II Pro Xilinx Multimedia and MicroBlaze board, we worked on six experiments that mainly involve programming the FPGA, learn how to use its peripherals, understand its internal architecture, study in details its internal connections and external peripherals as we would be dealing with the various bus protocols in order to be able to interconnect FPGAs using Ethernet as well as to be able to interconnect various processors on one single FPGA, as well as learn how to debug using the various debugging tools available.

The set of Experiments constitute the laboratory course **ECE 532_*Digital Hardware*** offered at the University of Toronto:

Lab Experiment 1

- Setting the FPGA ready to work.
- Creating a Base System and performing the hardware configuration (bitstream, netlist, internal memory configuration, I/O …)
- Defining the software and various compiler settings (compiler optimizations to the source code…)
- Using various debugging tools for verification.

Lab Experiment 2

- Using the OPB (*On Chip Peripheral Bus*).
- Using the OPB GPIO (*General Purpose I / O*) in order to control a User Led on the FPGA Board and understanding how the GPIO is memory mapped (so we control the Led by actually setting and resetting a specific bit in memory using the XMD (*Xilinx Multimedia Debugger*))
- Adding the software to use the GPIO (Device Driver to the GPIO which allows us to turn the led on and off)

Lab Experiment 3

- Improving the system built in the previous labs by including a peripheral input using DIP switches.
- Using the Timer / Counter peripheral device and connecting it to the MicroBlaze system using the OPB.
- Combining the Timer / Counter peripheral device with an Interrupt Controller module in order to generate continuous interrupts used for controlling the operation of input / output devices.

- Using and modifying existing Device Drivers to control operation of peripheral devices.

- Adding input to the system and modifying device drivers accordingly for control purpose via the GPIO bus which is memory mapped.

Lab Experiment 4

- Improving the system built in the previous labs by including a peripheral which is the EMAC (*Ethernet Medium Access Controller*) 10 / 100 Mbps module.

- Understanding how to configure the various options and parameters of EMAC controller such as the local MAC Address, automatic pad / removal of the 32 bit CRC (*Cyclic Redundancy Check*) ….

- Writing a small device driver to test the EMAC module. The device driver was configured for loopback operation where a frame is sent and received by the same module to test for various Ethernet fields' correctness as well as data consistency.

Lab Experiment 4. 5

- Introduction to the design of our own modules, in case functionality desired is not provided by Xilinx.

- Using a user defined module for profiling and collecting various statistics on the running program. Profiling using counters helps us determine which parts of the program should be placed on the faster On-Chip Block RAMs.

Lab Experiment 6

This experiment was an introduction to the Xilinx ISE 6, it consisted of the following topics:

- To gain an understanding of how to use ISE.

- To develop a simple core using ISE for use on the Xilinx Multimedia board.

- To use CoreGen IP (*Intellectual Property*) in this design.

- To learn how to initialize memory.

- To use iMPACT to download the design to the board.

- To use the pushbuttons on the Multimedia board.

Protocol Description

1. Introduction

In typical networking protocols, layering is the basis of design. Basically each layer has a specific role in the communication process and builds on top of the lower level layers. This provides protocols with modularity, scalability and facilitates the software / hardware interaction. For example in the TCP / IP stack, IP deals with end to end connectivity while TCP uses IP and builds on top of it in order to provide reliable process to process communication. UDP runs over IP, just like TCP, but instead keeps the communication simple and does not deal with reliability.

The Communication Protocol implemented in this project runs over Ethernet IEEE 802.3, and provides a higher level layer to provide reliable communication, as well as the capability to overcome payload size limitation per transfer as defined by the Ethernet protocol. So the protocol consists of two layers, the lower layer being Ethernet, and the higher layer, what we implemented in this project.

2. Layer 1 Overview: the Ethernet Protocol

In networking, Ethernet is referred to as a link layer protocol; it deals with medium access control. The Ethernet Payload exchanged is encapsulated in an Ethernet “frame”. Each Ethernet Frame contains a Header and a Trailer which contain relevant information for the delivery of the payload across a physical network, as well as to ensure that data exchanged is consistent.

The Xilinx Multimedia Board contains an Ethernet Controller and can be used for networking. The Ethernet Controller needs to be connected to a MicroBlaze system via an OPB and the setup of the controller has to be done using high level device driver functions. The controller can be configured in compliance with IEEE 802.3 specifications. An important note is that the controller contains a hard coded MAC Address in a register and can be obtained with a special procedure. However, the controller also has a configurable MAC Address that is used for user convenience. Moreover, some hardware features can be configured for user convenience such as automatic insertion and removal of some Ethernet Header and Trailer fields such as the CRC and the local MAC Address. Automatic insertion by hardware is done for performance purpose. For instance a CRC can have a custom hardware implementation that runs a lot faster than the equivalent sequential software implementation due to parallelization in hardware.

Ethernet Frame Format

Destination Address	Source Address	Frame Length	Payload	Frame Check Sequence (CRC)
6 bytes	**6 bytes**	**2 bytes**	**46 to 1500 bytes (automatic padding)**	**4 bytes (automatic insertion by hardware)**

The Source Address can be automatically inserted by the hardware. Padding occurs in the payload in case we need to transmit less than 46 bytes of effective payload. This is done to make collision detection possible in Ethernet based networks.

Some important notes about the Ethernet protocol justify the use of a higher level protocol for our system:

- *Flow Control*: In case a frame is lost, the Ethernet protocol does nothing to remedy with this problem. So the sender has no way of telling if the frame it sent was received correctly. Moreover in case a receiver gets two identical frames, it does not differentiate between them. Finally frames can be received out of order without clues for the receiver and data recovery would not be consistent without the help of a higher level protocol.
- *Payload Size*: A basic limitation of the Ethernet protocol is the size of the payload it can support. The upper limit on an Ethernet payload size is 1500 bytes. So we need a fragmentation / reassembly procedure in order to be able to send / receive messages of sizes larger than 1500 bytes.

3. Layer 2 Overview

Due to the reasons explained in the previous, a higher level protocol is needed. The higher level protocol defines a Layer 2 Header fields that is prepended to the actual payload of the message the user is sending. This layer 2 message, containing the layer 2 header and the layer 2 payload (actual user message) is then placed in the Ethernet payload field. At the receiver, the protocol reads the Ethernet payload field, and then extracts from it the various layer 2 header fields for further demultiplexing of the message.

Note that the layer 2 message lies in the Ethernet payload field.

Layer 2 Format

Frame Type	Sequence Number	Message Length	Fragment Payload
2 bytes	**4 bytes**	**4 bytes**	**Any size less than 2^{32} bytes**

Here is a brief description of the meaning and role of each of the layer 2 header fields:

- *Frame Type*: frames exchanged between the nodes may be of many types, for instance a node might send an acknowledgement frame to another node notifying it that it got the proper frame. Another frame might simply contain an actual message payload. In order to differentiate between these types of frames, we use a 16 bit type field of which we are currently using the first 3 bits, the rest being reserved for later use as well as for byte alignment of the header fields to facilitate memory access operation. This header field is also used to support fragmentation at the sender and reassembly at the receiver.

Frame Type Meanings

Frame Type	Meaning
0	**Non Fragmented Message**
1	**ACK Message**
2	**First Fragment**
3	**Internal Fragment**
4	**Last Fragment**

We should note that the maximum effective payload size per frame is 1490 bytes since 10 bytes are being taken by the layer 2 header.

- *Sequence Number*: The sequence number field in the layer 2 payload is used for many reasons. It ensures that a unique identifier between frames exists, which means that the receiver would always check a variable, called *waiting sequence number* from a certain node and will only accept a frame holding that sequence number. After accepting this frame it will increment this variable and wait for the next sequence number. This ensures that frames are always received in order, and that a same frame is never received twice by a node. Otherwise, without this flow control, many frames received out of order due to delays in switched for example, would cause demultiplexing problems for the receiver, especially in the case of fragmented messages. This field is also used in ACK Frames, it will then hold the sequence number of the frame that we are sending an ACK for. This ensures that the sender will exactly know that the frame was correctly received and that it can proceed with the next frame or simply notify the user about the success of the operation.

- *Message Length*: This field contains the total message length and is used for synchronization at the receiver.

An array of variables that is kept on each node, and is indexed by MAC Addresses, is used to update and read all the variables used for synchronizing senders and receivers.

a. Retransmission Schedule

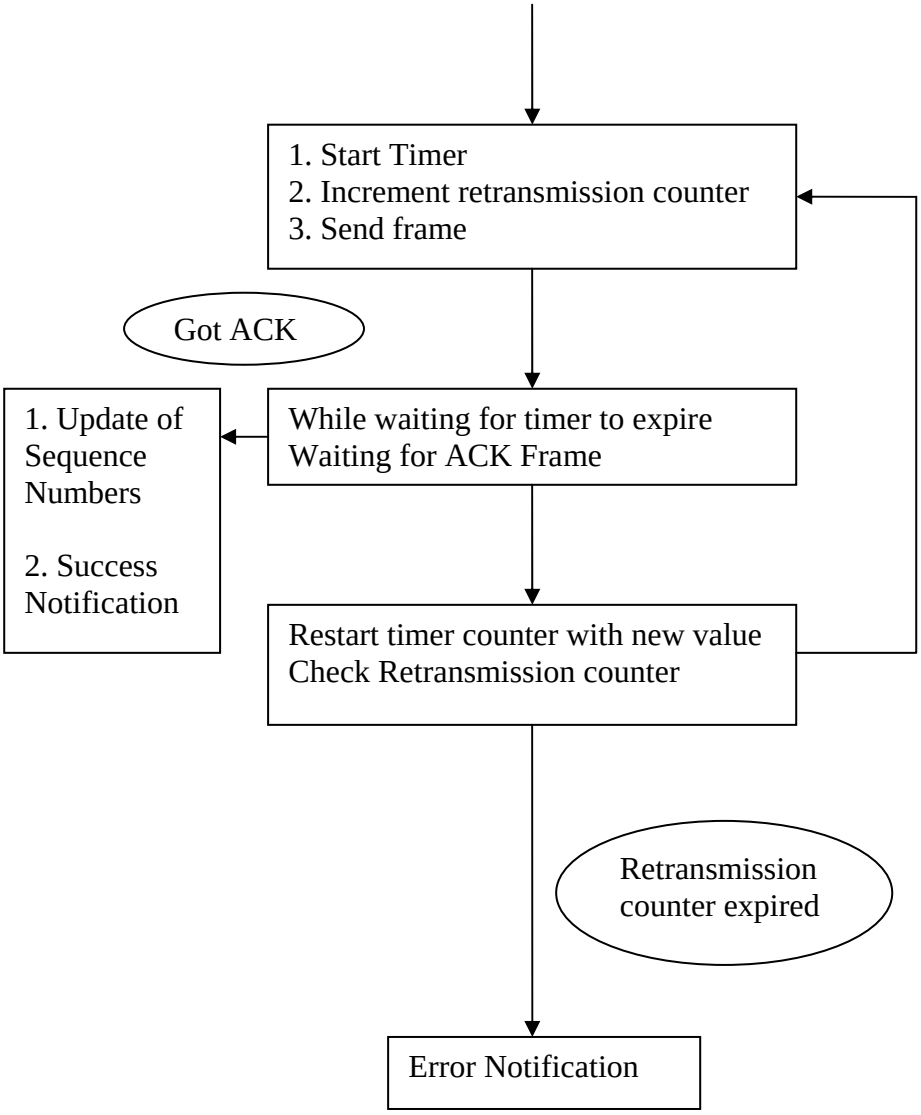
In networking, frames are lost due to excess collisions in hubs or buffer overflow in switches or routers or due to some non correctable bit corruption. That's why, the sender should always know if the receiver obtained the message correctly in order to know if it has to retransmit it or in some cases just abort sending the message. As explained before, the sequence numbers variables held by the nodes as well as the frame type layer 2 fields help us achieve this. Here is a description of the main retransmission schedule characteristics after a node attempts to send a message:

- A Frame is retransmitted 10 times before aborting and notifying the sender that the frame was not successfully sent.

- After each send attempt, a timer counter is reset and started in order to cause retransmission or abortion when the timer expires. Note that the timer counter is not always reset to the same value in order to create different delays, since networking theory expects such a scheme to work better due to changing network conditions and usability. Typically the waiting delays introduced gradually increase as the number of failures increase before reaching the maximum allowable retransmission attempts.

- When an appropriate Ack frame is received, the sender is notified of success and the appropriate sequence number is updated.

Here is a small diagram modelling the algorithm for the retransmission schedule for a single frame:



b. Fragmentation and Reassembly

Because Ethernet only supports a maximum of 1500 bytes of payload per frame, and because our layer 2 header fields occupy 10 bytes in every frame, we are constrained of having 1490 bytes of effective "user" data per frame. Messages can have random length and the protocol provides a segmentation and reassembly scheme to allow for transmission of message whose size is less than 2^{32} bytes or 4 GB. Here is a description of the characteristics of the segmentation / reassembly protocol:

- All messages of sizes less than or equal then 1490 bytes are not fragmented, and the type field in the layer 2 header has a value 0.

- Messages of sizes greater than 1490 bytes are fragmented and sent in order. The first fragment frame will contain the first 1490 bytes of the payload and will always be a type 2 frame. The last fragment frame will always be a type 4 frame. All other fragments, if they exist (in case the message needs more than two fragments), are of type three and will therefore contain 1490 byte payloads.

- The receiver reads the Layer 2 Message Length field as well as the message type to read the message fragmented payloads and allows the user to reassemble the payloads by having some data structure visible to the user.

c. Error Recovery Algorithms

This communication algorithm was designed to be light weight and efficient, while staying a general purpose multiprocessor communication algorithm. The sending and receiving sequence numbers array is very sensitive and should always stay consistent between the sender and the receiver. In fact, any inconsistency between two corresponding sequence numbers would lead to a communication link between two certain nodes be "cut". For example, if a certain node 'x' is expecting a sequence number frame from node 'y', and node 'y' thinks that node 'x' is expecting a different sequence number, node 'y' would always send frames that would be disregarded by node 'x'. That is, node 'x' would interpret those frames as if they were delayed frames that are out of order and would just reject them.

To avoid this problem of sequence number inconsistencies between two different nodes, we introduce some special checks in our algorithm:

- In case a node is waiting for frame 'x' and receives frame 'x-1' from a certain node, it will mean that the acknowledgement frame sent did not reach its destination, which caused the retransmission of the same frame. In that case, the same acknowledgment frame is just resent by the receiver to the sender to fix things.

- In the single MicroBlaze implementation, in case a certain frame of type 3 or 4 (internal fragment or last fragment) failed to reach the destination node, the sender would immediately know this because it would have expired and will reset the sequence number as it was before sending the first fragment (type 2 frame). The receiver will know that the whole message must be disregarded when it receives from a certain node a frame of type 0 or of type 2 (non fragmented message or first fragment) before getting the frame of type 4 (last fragment type). Then it will also reset the sequence number it was waiting for before it updated it when it got the type 2 frame.

System 1: Single MicroBlaze Communication System

Introduction

This system consists of one MicroBlaze system running on each multimedia board. The source code for the device drivers' configuration and protocol description runs on 64 KB on chip memory.

Hardware Configuration

This system consists of one MicroBlaze. The source code of **microblaze_0** runs on a 64KB on chip memory and occupies approximately 58 KB. In case the user needs to add code that will make the total code exceed the 64 KB limit, the system will have to be reconfigured to run from external ZBT Memories, which will have a certain negative effect on performance.

Peripheral	HW Ver	Instance	Base Address	High Address	Min
microblaze	2.10.a	microblaze_0			
opb_mdm	2.00.a	debug_module	0x80010000	0x800100ff	0x100
lmb_bram_if_cntlr	1.00.b	dlmb_cntlr	0x00000000	0x0000ffff	0x800
lmb_bram_if_cntlr	1.00.b	ilmb_cntlr	0x00000000	0x0000ffff	0x800
bram_block	1.00.a	lmb_bram			
opb_uartlite	1.00.b	RS232	0x80010100	0x800101ff	0x100
opb_gpio	3.01.a	opb_gpio_0	0x80010200	0x800103ff	0x1FF
opb_timer	1.00.b	opb_timer_0	0x80010400	0x800104ff	0x100
opb_intc	1.00.c	opb_intc_0	0x80010500	0x8001051f	0x20
opb_ethernet	1.00.m	opb_ethernet_0	0x80014000	0x80017fff	0x4000

- MicroBlaze_0: The processor.
- OPB: On Chip Peripheral Bus used to interconnect the processor to the various peripheral devices used.
- OPB GPIO: peripheral general purpose input output device connected to the MicroBlaze using the OPB, used to take input from a DIP Switch and drive output to a user LED.
- OPB MDM: peripheral Microprocessor debug module connected to the MicroBlaze using the OPB.
- OPB RS232: peripheral serial link device connected to the MicroBlaze using the OPB and used to drive the output serial link that is used to print statements for debugging and verification on a HyperTerminal on a computer.

- OPB ETHERNET: peripheral Ethernet controller device connected to the MicroBlaze using the OPB.

- OPB TIMER: peripheral timer counter device connected to the MicroBlaze using the OPB. This module is used to notify the sender that it has to retransmit a frame or abort.

- OPB INTC: peripheral interrupt controller device connected to the MicroBlaze using the OPB, and with the processor interrupt signal directly connected to the processor interrupt signal of the MicroBlaze. It takes as an input the interrupt output of the Ethernet controller (high priority) and the interrupt output of the timer counter (low priority).

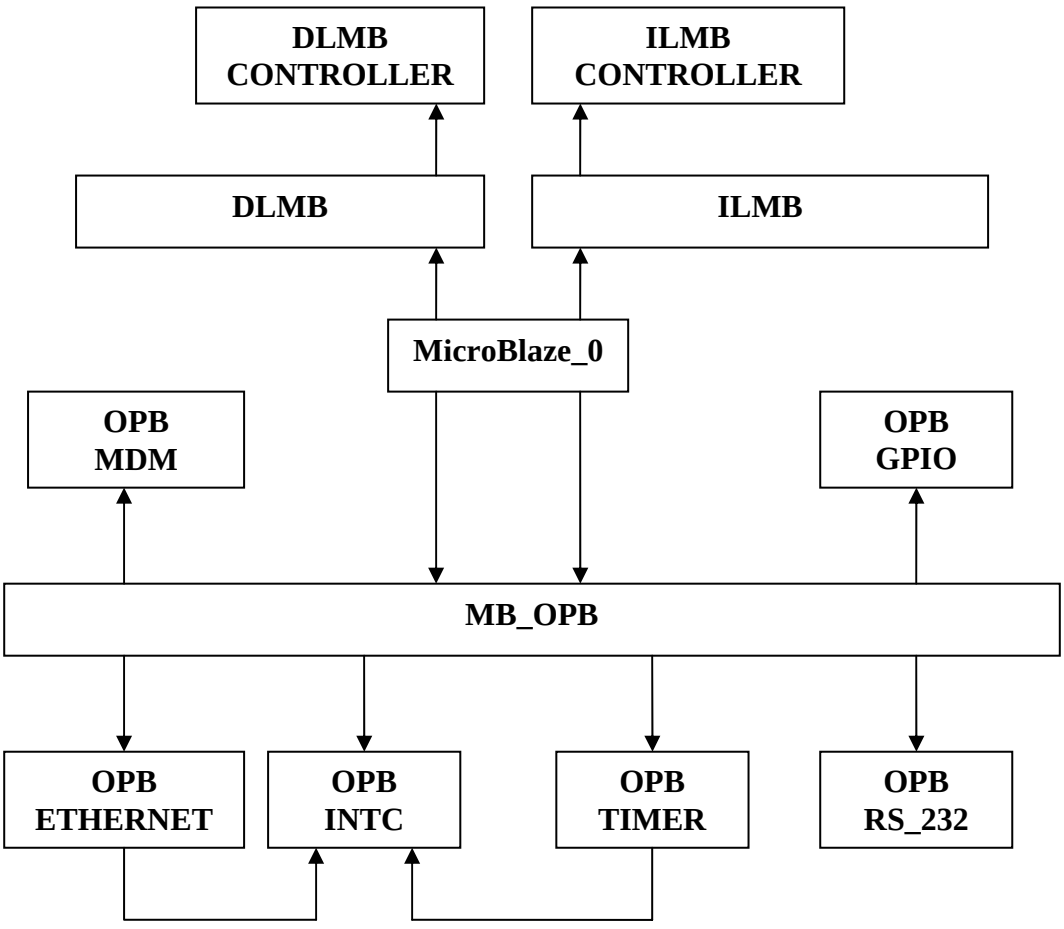
- DLMB: Data Local Memory Bus connected to the MicroBlaze.

- ILMB: Instruction Local Memory Bus connected to the MicroBlaze.

- DLMB CONTROLLER: Data Local Memory Bus Controller connected to DLMB.

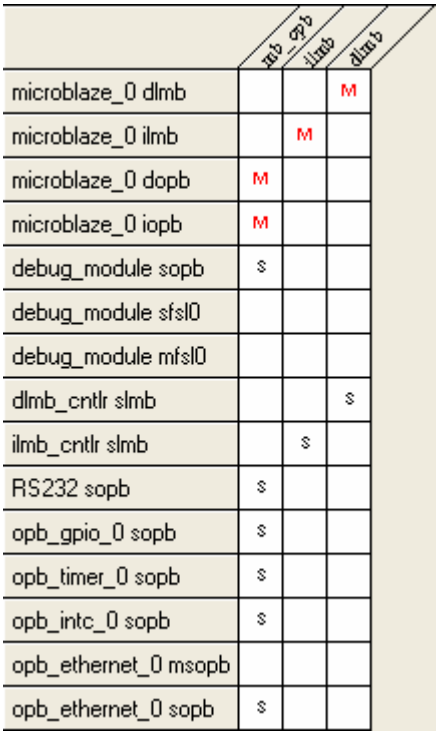
- ILMB CONTROLLER: Instruction Local Memory Bus Controller connected to ILMB.

Here is a schematic of the hardware modules in the system:



Bus Connections

Note that **M** means master and **s** means slave. For instance the opb_gpio_0 module is connected to the mb_opb bus as a slave. The MicroBlaze dopb and iopb are connected as masters to the mb_opb.



	mb_opb	ilmb	dlmb
microblaze_0 dlmb			M
microblaze_0 ilmb		M	
microblaze_0 dopb	M		
microblaze_0 iopb	M		
debug_module sopb	S		
debug_module sfsl0			
debug_module mfsl0			
dlmb_cntlr slmb			S
ilmb_cntlr slmb		S	
RS232 sopb	S		
opb_gpio_0 sopb	S		
opb_timer_0 sopb	S		
opb_intc_0 sopb	S		
opb_ethernet_0 msopb			
opb_ethernet_0 sopb	S		

Ports

The input signal **Intr** of the interrupt controller consists of **timer_intr & emac_intr**. Effectively this will give higher interrupt priority to the interrupt output **emac_intr** of the emac module, and a lower priority to the interrupt output **timer_intr** of the timer counter module. The output **IntConnection2** of the interrupt controller is connected to the interrupt input of the MicroBlaze.

All the clocks in the system are synchronized to **sys_clk** clock signal that is connected externally via the .ucf file to the board's crystal oscillator.

The input and output of the Gpio module are also connected to external connections which are the DIP Switch and the User Led. All ports connected with **external scope** are connected to external pins. The description of these connections is in the .ucf file.

Instance	Port Name	Net Name	Pola...	Scope	Range	Class	Sensitivity
system	sys_clk	sys_clk_s	input	External		CLK	
system	sys_rst	sys_rst_s	input	External			
system	RS232_req_to_send	net_gnd	OUT...	External			
system	PHY_slew2	net_vcc	OUT	External			
system	PHY_slew1	net_vcc	OUT	External			
opb_timer_0	Interrupt	timer_intr	O	Internal		INTERRUPT	LEVEL_HIGH
opb_intc_0	Intr	timer_intr & emac_intr	I	Internal		INTERRUPT	
opb_intc_0	Irq	IntConnection2	O	Internal		INTERRUPT	
opb_gpio_0	GPIO_IO	opb_gpio_0_GPIO_IO	IO	External	[0:1]		
opb_ethernet_0	PHY_tx_data	opb_ethernet_0_PHY_tx_data	O	External	[3:0]		
opb_ethernet_0	IP2INTC_Irpt	emac_intr	O	Internal		INTERRUPT	LEVEL_HIGH
opb_ethernet_0	PHY_tx_er	opb_ethernet_0_PHY_tx_er	O	External			
opb_ethernet_0	PHY_crs	opb_ethernet_0_PHY_crs	I	External			
opb_ethernet_0	PHY_dv	opb_ethernet_0_PHY_dv	I	External			
opb_ethernet_0	PHY_tx_clk	opb_ethernet_0_PHY_tx_clk	I	External			
opb_ethernet_0	PHY_rx_er	opb_ethernet_0_PHY_rx_er	I	External			
opb_ethernet_0	PHY_rx_data	opb_ethernet_0_PHY_rx_data	I	External	[3:0]		
opb_ethernet_0	PHY_col	opb_ethernet_0_PHY_col	I	External			
opb_ethernet_0	PHY_rx_clk	opb_ethernet_0_PHY_rx_clk	I	External			
opb_ethernet_0	PHY_tx_en	opb_ethernet_0_PHY_tx_en	O	External			
opb_ethernet_0	PHY_Mii_clk	opb_ethernet_0_PHY_Mii_clk	IO	External			
opb_ethernet_0	PHY_Mii_data	opb_ethernet_0_PHY_Mii_data	IO	External			
microblaze_0	INTERRUPT	IntConnection2	I	Internal		INTERRUPT	LEVEL_HIGH
microblaze_0	CLK	sys_clk_s	I	Internal		CLK	
mb_opb	SYS_Rst	sys_rst_s	I	Internal			
mb_opb	OPB_Clk	sys_clk_s	I	Internal		CLK	
ilmb	LMB_Clk	sys_clk_s	I	Internal		CLK	
ilmb	SYS_Rst	sys_rst_s	I	Internal			
dlmb	SYS_Rst	sys_rst_s	I	Internal			
dlmb	LMB_Clk	sys_clk_s	I	Internal		CLK	
debug_module	OPB_Clk	sys_clk_s	I	Internal		CLK	
RS232	TX	RS232_TX	O	External			
RS232	OPB_Clk	sys_clk_s	I	Internal		CLK	
RS232	RX	RS232_RX	I	External			

External Port Connections: .UCF File

Net sys_clk PERIOD = 37037 ps;
Net sys_clk LOC=AH15;
Net sys_rst LOC=AH7;

Net RS232_RX LOC=C8;
Net RS232_TX LOC=C9;
Net RS232_req_to_send LOC=B8;

Net opb_gpio_0_GPIO_IO<1> LOC=B27;
Net opb_gpio_0_GPIO_IO<0> LOC=D10;

Net PHY_slew1 LOC=G16;
Net PHY_slew2 LOC=C16;
Net opb_ethernet_0_PHY_crs LOC=F20;
Net opb_ethernet_0_PHY_col LOC=C23;
Net opb_ethernet_0_PHY_tx_data<3> LOC=C22;
Net opb_ethernet_0_PHY_tx_data<2> LOC=B20;
Net opb_ethernet_0_PHY_tx_data<1> LOC=B21;
Net opb_ethernet_0_PHY_tx_data<0> LOC=G20;

```
Net opb_ethernet_0_PHY_tx_en LOC=G19;
Net opb_ethernet_0_PHY_tx_clk LOC=H16;
Net opb_ethernet_0_PHY_tx_er LOC=D21;
Net opb_ethernet_0_PHY_rx_er LOC=D22;
Net opb_ethernet_0_PHY_rx_clk LOC=C17;
Net opb_ethernet_0_PHY_dv LOC=B17;
Net opb_ethernet_0_PHY_rx_data<0> LOC=B16;
Net opb_ethernet_0_PHY_rx_data<1> LOC=F17;
Net opb_ethernet_0_PHY_rx_data<2> LOC=F16;
Net opb_ethernet_0_PHY_rx_data<3> LOC=D16;
Net opb_ethernet_0_PHY_Mii_clk LOC=D17;
Net opb_ethernet_0_PHY_Mii_data LOC=A17;
```

Using the Single Microblaze Version of the Communication Protocol and Network Monitoring using a Sniffer Node

Introduction*:*

In this small experiment, you will be introduced on how to configure two boards for communication.

- Selection of MAC Addresses for each board.
- At the Sender: Using the Send function and verifying the success or failure of the operation.
- At the Receiver: Using the receive handler function and some global variables to demultiplex the incoming frame payload.
- Using the Sniffer: A third board will be used as a sniffer to monitor network transactions.

Preparation*:*

Read the documentation on the single MicroBlaze System as well as that on the protocol description before you start with the setup.

Equipment Needed*:*

- Two Multimedia Boards, or three in case you want to use a sniffer node.
- A Hub or a Switch and appropriate Ethernet RJ Cables. Note that you should pay attention of *not* using crossed cables! In case you are using a sniffer node, you should use a hub.

Setup*:*

1. In case you want to use a sniffer (Recommended for this experiment), read the document “Setting up the Sniffer Node” and perform the steps listed under *setup.* After completing this step, the sniffer should be operational on the network.

2. Go to your **`SINGLE_MB_EMAC`** Directory. Then go to the **`APP1`** directory. In the **`APP1`** directory double click on the XPS Icon.

3. Under Application Tab, expand the *Sources* Submenu of microblaze_0 and open the source code. Every time you will toggle the DIP Switch SW0 you will be sending a frame to the other board.

4. Take a look at the SingleMB_MACAddresses.h file in the **code** directory to know the local address of this node, and view the address definitions on this network and various function and global variable definitions.

5. Now take a look at SingleMB_MACAddresses.c file in the **code** directory. You will need to modify the **GetIndex** function as well as some global variables and add address definitions in the header file in case you are expanding the network.

6. Now take a look at the *FifoRecvHandler* in the SingleMB_HandlerFunctions.c file under the code directory of **`APP1`**. You will notice regions under *User Code.* A Small test user code has been written to show the variables to use to demultiplex the frame’s payload into the appropriate user’s buffer.

7. Repeat steps 2-6 for **`APP2`**.

8. Setup a Terminal Window for the Serial Port.

9. Build the applications, update the bitstreams, and download them (the program is downloaded as an executable)

10. After downloading the two programs on the two Multimedia Boards toggle DIP Switch a few times on each board to see what happens (you should have the serial port connected to either one). Now toggle the DIP Switch on the second board and see what happens on the Terminal window. Move the serial port to the sniffer node and toggle the DIP Switch on either board to see what happens on the HyperTerminal screen, you will be seing either ACK Frames or Message Frames.

11. Try to make some modifications to the main section of the program and redo this experiment few times to get familiar with the software.

Software Configuration and Source Code

Test1.c

```
/*****************************************************************************
*
*       Simple Reliable Communication System over Ethernet _ Single MicroBlaze System
*
*       A simple DIP switch is used to send frames
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher-John Comis (University of Toronto)
*       at the following email address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/

/*****************************  Include Files *********************************/

#include "xemac.h"                      // Ethernet Module
#include "xparameters.h"
#include "xstatus.h"
```

```
#include "mb_interface.h"          // MicroBlaze
#include "xintc.h"                 // for interrupt control
#include "xgpio.h"                 // for the switch
#include "xtmrctr.h"               // To implement delayed frame retransmissions

#include "SingleMB_MACAddresses.h"              // contains the MAC Address Definitions and some functions
#include "SingleMB_GlobalVariables.h"           // contains the global variables shared among the 5 files
#include "SingleMB_HandlerFunctions.h"          // contains handler functions
#include "SingleMB_SendFunctions.h"             // contains the send functions of frame and message layers
#include "SingleMB_InitializationFunctions.h"   // contains the initialization functions
#include "SingleMB_DebuggingFunctions.h"        // contains the functions to display frame
                                                // and message headers and payload...

#include "SingleMB_MACAddresses.c"              // contains the MAC Address Definitions and some functions
#include "SingleMB_HandlerFunctions.c"          // contains handler functions
#include "SingleMB_SendFunctions.c"             // contains the send functions of frame and message layers
#include "SingleMB_InitializationFunctions.c"   // contains the initialization functions
#include "SingleMB_DebuggingFunctions.c"        // contains the functions to display frame
                                                // and message headers and payload...

/*********************************  MAIN FUNCTION  ******************************/

int main()
{
        /* Definitions */

        Xuint32 counter;
        Xuint32 debouncing;
        Xuint8 * BigPtr;
```

```
	Xuint8 * TempAddressPtr;

	/* Initializations */

	Initialize_Parameters();		// for EMAC and TIMERCOUNTER and INTERRUPTS and GPIO

	/*		*/

	BigPtr = (Xuint8 *)BigBuffer;
	for(counter = 0 ; counter < 4400 ; counter++)   // fill about two and a half buffers
	{
		*(Xuint8 *) BigPtr = (Xuint8) (counter % 100)	;
		BigPtr ++;
	}
	BigPtr = (Xuint8 *)BigBuffer;


while(1)
{
	if(XGpio_DiscreteRead(&Gpio) >= 2  && reload_flag == 1) // input DIP switch is on
	{
			for(debouncing = 0 ; debouncing < 250 ; debouncing++);
			if(XGpio_DiscreteRead(&Gpio) >= 2  &&  reload_flag == 1)
			{
				if(Fragment_and_Send_Reliably(1510, BigPtr, Address2) == XST_SUCCESS)
				{
					xil_printf("Sending Succeeded !!\n\r");
				}
				else
				{
```

```
                                        xil_printf("Sending Error !!\n\r");
                                }

                                reload_flag = 0; // need to toggle DIP switch SW2
                        }
                }
        else if (XGpio_DiscreteRead(&Gpio) < 2 && reload_flag != 1 )
                {
                        for(debouncing = 0 ; debouncing < 250 ; debouncing++);

                        if(XGpio_DiscreteRead(&Gpio)<2 && reload_flag !=1)
                        {
                                xil_printf("Reloading !!\n\r");

                                reload_flag=1;          // now we can resend
                        }
                }
   }                                                    // end of while(1)


        /*************************
        ** User Code Ends Here **
        *************************/

        while(1)
        {
        }

        return 0;
}
```

```
/*****************************************************************************************/
```

SingleMB_GlobalVariables.h

```
/*****************************************************************************
*
*       Simple Reliable Communication System over Ethernet _ Single MicroBlaze System
*
*       Global Variables Definitions
*
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher-John Comis (University of Toronto)
*       at the following email address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/
```

```
#ifndef SINGLEMB_GLOBALVARIABLES_H
#define SINGLEMB_GLOBALVARIABLES_H

#include "SingleMB_MACAddresses.h"

/***************************** Constants Definitions ***********************************/

#define XEM_MAX_FRAME_SIZE_IN_WORDS ((XEM_MAX_FRAME_SIZE / sizeof(Xuint32)) + 1)


#define TEST_1_OPTIONS              (XEM_UNICAST_OPTION | XEM_INSERT_PAD_OPTION | \
                                    XEM_INSERT_FCS_OPTION | XEM_INSERT_ADDR_OPTION | \
                                    XEM_OVWRT_ADDR_OPTION)


#define LED   0x1                                   // bit 0 of GPIO is connected to the LED

#define INTC_INPUT_EMAC             0               // EMAC has higher priority (check add/edit cores)
#define INTC_INPUT_TIMER        1                   // TIMER has lower priority

#define TIMER_COUNTER_0     0           // Timer Counter 0 is used
#define INTC_DEVICE_ID          0


#define ACK_L2_PAYLOAD_SIZE         36
#define ACK_L1_PAYLOAD_SIZE         46
#define ACK_FRAME_SIZE              60
#define L2_HEADER_SIZE              10
#define MAX_RETRANSMIT_ATTEMPTS         10
```

```
#define ACK_MESSAGE_TYPE                1
#define NON_FRAGMENTED_MESSAGE_TYPE     0
#define FIRST_FRAGMENT_TYPE             2
#define INTERNAL_FRAGMENT_TYPE          3
#define LAST_FRAGMENT_TYPE              4
#define MAX_L2_PAYLOAD                      1490


/*************************  Instance Definitions  ************************************/

static XGpio Gpio;
static XEmac Emac;
static XIntc InterruptController;
static XTmrCtr TimerCounter;

XEmac *EmacPtr;                                 // Initialized in Init.h



/*************************  Buffer Variables  ******************************/

static Xuint32 TemporaryBuffer;      // To hold temporarily 32 bit values for alignment problems

static Xuint32 TxMessage[XEM_MAX_FRAME_SIZE_IN_WORDS + 40]; // holds fragment to send
static Xuint32 TxFrame[XEM_MAX_FRAME_SIZE_IN_WORDS];            // holds frame to send

static Xuint32 RecvBuffer[XEM_MAX_FRAME_SIZE_IN_WORDS];         // holds incoming data
static Xuint32 RecvPayload[XEM_MAX_FRAME_SIZE_IN_WORDS];  // holds incoming payload
```

```
static Xuint32 BigBuffer[XEM_MAX_FRAME_SIZE_IN_WORDS * 3];  // holds message to send

static Xuint8 AckFrame[60];                                                    // holds the ACK to send


/*************************  Global Variables    ******************************/

char array[17];                              // used to hold MAC addresses in standard format


Xuint32 RstVal[10] = {  0xFE600000,0xFF000000,0xFF000000,
                                        0xFE000000 ,0xFE000000 ,0xFE000000 , 0xFD500000,
                                        0xFD000000, 0xFC700000 , 0xFC000000};

// RESET VALUE --> DELAY IN SECONDS
// 0xF0000000  --> 10 seconds approx.
// 0xF7000000  --> 5.5 seconds approx.
// 0xFE600000  --> 1 second approx.
// 0xFF400000  --> 0.5 seconds approx.



/************************* Acknowledgement Variables ***********************/

/*
*
* This structure contains all the variables to be used between
* Two Specific Nodes.
*
* This type of implementation allows us to have seperate reliable links
```

```
*
* For a system of 'N' Communication Nodes, each single node will have 'n - 1' structures,
* each one for communicating with one neighbor.
*
* This allows gracefull degradation of the system. If a shared communication variable between node
* x and node y becomes invalid (each of x and y have different copies of this variable), even though this
* is unlikely to happen due to the fact that the system is designed with an error recovery algorithm,
* only the link between x and y will be broken, while x and y would continue to talk to the rest of the
* multiprocessor nodes.
*
*/

XEmac_Stats Stats;  // One Variable for the EMAC Module.


Xuint8 reload_flag;                 // Changes with DIP Switch Input_this is only used in the main to manually send frames
Xuint8 Timer_Expired;               // turns to one everytime the timer expires
                                    // We only have one timer in the system


typedef struct                  /* Comm Vars Structure*/
{

Xuint32 SeqNum;                 // Holds Wrapping Off Sequence Numbers of Messages

Xuint32 WaitingFrameNumber;     // Holds the Frame Seq Num we are waiting for

Xuint8 Ack_Received;            // turns to one in case we got an ACK Message

Xuint8 ExpectMore;              // used to have proper reception of Fragmented Messages
```

```
Xuint32 RememberSeqNumReceiver;        // used to remember the sequence number of the starting sequence at the receiver

Xuint32 NbFragRecv;                    // contains the previous number of fragments as part of a same message

Xuint32 MsgLength;

Xuint32 FragLength;

}CommunicationsVariables;


CommunicationsVariables CommVars[5];
                        // holds 5 sets of variables for comm between board 0 to 4 in MAC terms

                        // MAC Addresses are :
                        //
                        // 0  --> 06 05 04 03 02 00
                        // 1  --> 06 05 04 03 02 01
                        // 2  --> 06 05 04 03 02 02
                        // 3  --> 06 05 04 03 02 03
                        // 4  --> 06 05 04 03 02 04
                        //

#endif
```

SingleMB_DebuggingFunctions.h

```
/*****************************************************************************
*
*       Simple Reliable Communication System over Ethernet _ Single MicroBlaze System
*
*       Definitions of some functions for debugging and displaying frame content
*       and headers
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher-John Comis (University of Toronto)
*       at the following email address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/

#ifndef SINGLEMB_DEBUGGINGFUNCTIONS_H
#define SINGLEMB_DEBUGGINGFUNCTIONS_H

#include "SingleMB_GlobalVariables.h"

/*
*   returns the character equivalent of an integer
```

```
*/
char map(int x);


/*
* Pass it a Pointer to the start of the Ethernet Header
* Returns the Layer 2 Type Field
*/
Xuint16 GetType(Xuint8 *MsgBuffer);

/*
* Pass it a Pointer to the start of the Ethernet Header
* Returns the Layer 2 Sequence Number Field
*/
Xuint32 GetSeqNum(Xuint8 *MsgBuffer);

/*
* Pass it a Pointer to the start of the Ethernet Header
* Returns the Layer 2 Message Size Field
*/
Xuint32 GetSize(Xuint8 *MsgBuffer);

/*
* Pass it a pointer to the EMAC Address
* Fills the global variable 'array' with
* the ETHERNET Address in standard format
*/
void fill_array(Xuint8 *Ptr);

/*
```

```
* Prints the EMAC Statistics
*/
void PrintEmacStats(XEmac_Stats Stats);

/*
* Pass it a pointer to the Ethernet Frame
*
* Prints the Layer 2 Variables
*
* Message Header Format:     Type                    _ 2 bytes
*                                                    Sequence Number       _ 4 bytes
*                       Message Length      _ 4 bytes
*/
void Print_Message_Fields(Xuint8 *MsgBuffer, int MsgLen);

/*
* Pass it a pointer to the Ethernet frame
*
* and the MsgLen as returned by the Receive function
*
* Ethernet Header Format:     Destination Address _ 6 bytes
*                                                    Source Address      _ 6 bytes
*                       Len / Type Field    _ 2 bytes
*/
void Print_Frame_Fields(Xuint8 *MsgBuffer, int MsgLen);

#endif
```

SingleMB_DebuggingFunctions.c

```
/*****************************************************************************
*
*       Simple Reliable Communication System over Ethernet _ Single MicroBlaze System
*
*       Source Code of functions for debugging and displaying frame content
*       and headers
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher-John Comis (University of Toronto)
*       at the following email address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/


#include "SingleMB_DebuggingFunctions.h"


/*
*   returns the character equivalent of an integer
*/
char map(int x)
```

```
{
        switch(x)
        {
        case 0:
                return '0';
        case 1:
                return '1';
        case 2:
                return '2';
        case 3:
                return '3';
        case 4:
                return '4';
        case 5:
                return '5';
        case 6:
                return '6';
        case 7:
                return '7';
        case 8:
                return '8';
        case 9:
                return '9';
        case 10:
                return 'A';
        case 11:
                return 'B';
        case 12:
                return 'C';
        case 13:
```

```
                return 'D';
        case 14:
                return 'E';
        case 15:
                return 'F';
        }
}


/*
* Pass it a Pointer to the start of the Ethernet Header
* Returns the Layer 2 Type Field
*/
Xuint16 GetType(Xuint8 *MsgBuffer)
{
        Xuint8 * Ptr = MsgBuffer;
        Ptr+=14;
        return *(Xuint16 *)Ptr;
}

/*
* Pass it a Pointer to the start of the Ethernet Header
* Returns the Layer 2 Sequence Number Field
*/
Xuint32 GetSeqNum(Xuint8 *MsgBuffer)
{
        Xuint8 * Ptr = MsgBuffer;
        Ptr+=16;
        return *(Xuint32 *)Ptr;
}
```

```
/*
* Pass it a Pointer to the start of the Ethernet Header
* Returns the Layer 2 Message Size Field
*/
Xuint32 GetSize(Xuint8 *MsgBuffer)
{
        Xuint8 * Ptr = MsgBuffer;
        Ptr+=20;
        return *(Xuint32 *)Ptr;
}


/*
* Pass it a pointer to the EMAC Address
* Fills the global variable 'array' with
* the ETHERNET Address in standard format
*/
void fill_array(Xuint8 *Ptr)
{
        Xuint8 mycounter;
        for(mycounter = 0 ; mycounter < 17 ; mycounter += 3)
        {
                array[mycounter+1] = map(*(Ptr) & 0xF);
                array[mycounter] = map( ( *(Ptr) & 0xF0 ) >> 4);
                Ptr++;
                if(mycounter<15) array[mycounter+2] = '-';
        }
}
```

```
/*
* Prints the EMAC Statistics
*/
void PrintEmacStats(XEmac_Stats Stats)                                          // for debugging
{
   xil_printf("Number of frames transmitted                  %d\n\r", Stats.XmitFrames);
//   xil_printf("Number of bytes transmitted                  %d\n\r", Stats.XmitBytes);
   xil_printf("Number of transmission failures due to late collisions   %d\n\r", Stats.XmitLateCollisionErrors);
   xil_printf("Number of transmission failures due to excess collision deferrals   %d\n\r", Stats.XmitExcessDeferral);
//   xil_printf("Number of transmit overrun errors   %d\n\r", Stats.XmitOverrunErrors);
//   xil_printf("Number of transmit underrun errors  %d\n\r", Stats.XmitUnderrunErrors);
   xil_printf("Number of frames received              %d\n\r", Stats.RecvFrames);
//   xil_printf("Number of bytes received                  %d\n\r", Stats.RecvBytes);
   xil_printf("Number of frames discarded due to FCS errors                %d\n\r", Stats.RecvFcsErrors);
   xil_printf("Number of frames received with alignment errors      %d\n\r", Stats.RecvAlignmentErrors);
//   xil_printf("Number of frames discarded due to overrun errors   %d\n\r", Stats.RecvOverrunErrors);
//   xil_printf("Number of recv underrun errors       %d\n\r", Stats.RecvUnderrunErrors);
//   xil_printf("Number of frames missed by MAC   %d\n\r", Stats.RecvMissedFrameErrors);
   xil_printf("Number of frames discarded due to collisions          %d\n\r", Stats.RecvCollisionErrors);
   xil_printf("Number of frames discarded with invalid length field   %d\n\r", Stats.RecvLengthFieldErrors);
//   xil_printf("Number of short frames discarded     %d\n\r", Stats.RecvShortErrors);
//   xil_printf("Number of long frames discarded      %d\n\r", Stats.RecvLongErrors);
//   xil_printf("Number of DMA errors since init      %d\n\r", Stats.DmaErrors);
//   xil_printf("Number of FIFO errors since init      %d\n\r", Stats.FifoErrors);
//   xil_printf("Number of receive interrupts                %d\n\r", Stats.RecvInterrupts);
//   xil_printf("Number of transmit interrupts          %d\n\r", Stats.XmitInterrupts);
//   xil_printf("Number of MAC (device) interrupts   %d\n\r", Stats.EmacInterrupts);
//   xil_printf("Total interrupts                                %d\n\r", Stats.TotalIntrs);
}
```

```
/*
* Pass it a pointer to the Ethernet Frame
*
* Prints the Layer 2 Variables
*
* Message Header Format:    Type                    _ 2 bytes
*                                                   Sequence Number      _ 4 bytes
*                     Message Length     _ 4 bytes
*/
void Print_Message_Fields(Xuint8 *MsgBuffer, int MsgLen)
{
        Xuint32 mycounter;
        Xuint32 LenField;
        Xuint16 TypeofMess;
        Xuint32 FragmentPayloadLength;          // since FragmentLength can be different
                                                // than Message PayloadLength
        Xuint8 *MesgPtr = MsgBuffer;
        Xuint8 *TemporaryPtr = (Xuint8 *)TemporaryBuffer;

        xil_printf("Message Parameters\r\n");
        xil_printf("_____________________\r\n");

        MesgPtr+=14;
        TypeofMess = *(Xuint16 *)MesgPtr;
        xil_printf("Message Type Field              : %d\n\r", TypeofMess);

        MesgPtr+=2;

        xil_printf("Message Sequence Nb Field       : %d\n\r", *(Xuint32 *)MesgPtr);
```

```
MesgPtr+=4;
LenField = *(Xuint32 *)MesgPtr;
xil_printf("Message Length                    : %d\n\r", LenField);

MesgPtr+=4;

if(LenField <= 1490)
        FragmentPayloadLength = LenField;          // The Frame contains the Message header (10 bytes)
                                                   // and the whole message itself as its payload
  // case of fragmented message
else
{
        if(TypeofMess == FIRST_FRAGMENT_TYPE || TypeofMess == INTERNAL_FRAGMENT_TYPE) // frame is full
                FragmentPayloadLength = MAX_L2_PAYLOAD;
        else if (TypeofMess == LAST_FRAGMENT_TYPE)         // last fragment of a message
        {
                if(LenField % MAX_L2_PAYLOAD == 0)         // 1490 payload size of last fragment
                        FragmentPayloadLength = MAX_L2_PAYLOAD;
                else
                        FragmentPayloadLength = (LenField % MAX_L2_PAYLOAD);
        }
}

        // Printing the first byte of the fragment payload
xil_printf("Fragment Payload 1st byte: %x\n\r", *(Xuint8 *)MesgPtr);

        // Getting to the last byte of the payload
MesgPtr += (FragmentPayloadLength - 1);

        // Printing the last byte of the fragment payload
```

```
	xil_printf("Fragment Payload last byte: %x\n\r", *(Xuint8 *)MesgPtr);

}

/*
* Pass it a pointer to the Ethernet frame
*
* and the MsgLen as returned by the Receive function
*
* Ethernet Header Format:	Destination Address _ 6 bytes
*                                                 Source Address      _ 6 bytes
*                       Len / Type Field    _ 2 bytes
*/
void Print_Frame_Fields(Xuint8 *MsgBuffer, int MsgLen)
{
	Xuint8 *MesgPtr = MsgBuffer;

	xil_printf("Frame Parameters\r\n");
	xil_printf("_________________\r\n");

	xil_printf("Frame Length                    : %d\r\n", MsgLen);

	fill_array(MesgPtr);
	xil_printf("Destination MAC                 : %s\r\n", array);

	MesgPtr+=6;

	fill_array(MesgPtr);
	xil_printf("Source      MAC                 : %s\r\n", array);
```

```
	MesgPtr+=6;

	xil_printf("Payload Length			: %d\r\n", *(Xuint16 *)MesgPtr);
}
```

SingleMB_SendFunctions.h

```
/*****************************************************************************
*
*	Simple Reliable Communication System over Ethernet _ Single MicroBlaze System
*
*	Send Functions Definitions
*
*	Author:		Patrick E. Akl
*			American University of Beirut
*			August 2004
*
*	For any comments, suggestions or bug discovery, please contact me at the following
*	email address : patrickakl@hotmail.com
*
*	or contact Christopher-John Comis (University of Toronto)
*	at the following email address:
*					comis@eecg.toronto.edu
*
*****************************************************************************/

#ifndef SINGLEMB_SENDFUNCTIONS_H
#define SINGLEMB_SENDFUNCTIONS_H
```

```
#include "SingleMB_GlobalVariables.h"
#include "SingleMB_DebuggingFunctions.h"

/*
* Fills the Ethernet Header and the Payload into TxFrame Buffer
* Source Address automatically included by PHY
*
* returns XST_SUCCESS or XST_FAILURE for just sending the frame
*
* Further reliability should be implemented seperatly
*
* Ethernet Header Format:     Destination Address _ 6 bytes
*                                                 Source Address      _ 6 bytes
*                         Len / Type Field    _ 2 bytes
*/
static XStatus SendFrame(XEmac *EmacPtr, Xuint32 FramePayloadSize, Xuint8* InputBuffer,
                         Xuint8 *DestAddress);

/*
* Pass it a pointer to the message we want to send an ACK for
*
* Destination MAC address for the ACK determined automatically
*
* NOTE: this will send an ACK only for frames that are non ACK Messages
*       The check for this is in the 'if' statement
*
* Returns XST_SUCCESS or XST_FAILURE
*
* XST_FAILURE      if we failed to send
*                                 or if we tried to send an ACK for an ACK
```

```
*/
XStatus Send_Ack_Message(Xuint8 *MsgBuffer);

/*
* Fills the Layer 2 Header and the Payload into TxMessage Buffer
* It calls the SendFrame to fill the Ethernet header and send the frame
*
* Note: the messages sent by this function can only be <= 1490 bytes in length
* returns XST_SUCCESS or XST_FAILURE for just sending the frame
*
* Further reliability should be implemented seperatly
*
*  Message Header Format:    Type                    _ 2 bytes
*                                                    Sequence Number       _ 4 bytes
*                      Message Length      _ 4 bytes
*/
static XStatus SendMessage(Xuint32 MessagePayloadSz, Xuint8* InputBuffer,
                    Xuint8 *DestAddress,  Xuint16 TypeofMessage);


/*
* Uses send message to send the payload passed through the pointer InputBuffer
*
* tried to send it 10 times after which it returns XST_FAILURE
* returns XST_SUCCESS if an ACK is received for this message
*
* Uses different waiting times for resetting the timer.
* Once the timer expires it sets Timer_Expired = 1
*
* Ack_Received is set to 1 by the receive handler function
```

```
*/
static XStatus SendReliably(Xuint32 MessagePayloadSz, Xuint8 *InputBuffer, Xuint8 *DestAddress, Xuint16 TypeofMessage);


/*
* Fragments the content pointed to by input buffer and sends the fragmented message across successive
* maximum payload 1490 bytes frames.
*
* In case one of the messages failed to arrive reliably to the receiver,
* the sequence numbers are reset so what they were before starting to send the message
*
* a very unlikely deadlock case is if the last ack that should be received by this sender failed to reach
* us 10 times. In this case, the sender would reset its sequence numbers, whereas the receiver would have the waiting sequence numbers
* after the last received message.
*
* Future Work: Deal properly with this case. Maybe send some special type of message to have like a sequence number handshake
* or agreement.
*/
static XStatus Fragment_and_Send_Reliably(Xuint32 MessagePayloadSz, Xuint8 *InputBuffer, Xuint8 *DestAddress);


#endif
```

SingleMB_SendFunctions.c

```
/*****************************************************************************
*
*       Simple Reliable Communication System over Ethernet _ Single MicroBlaze System
*
*       Source Code for the Send Functions
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher-John Comis (University of Toronto)
*       at the following email address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/

#include "SingleMB_SendFunctions.h"


/*
* Fills the Ethernet Header and the Payload into TxFrame Buffer
* Source Address automatically included by PHY
*
* returns XST_SUCCESS or XST_FAILURE for just sending the frame
```

```
*
* Further reliability should be implemented seperatly
*
* Ethernet Header Format:     Destination Address _ 6 bytes
*                                                 Source Address      _ 6 bytes
*                         Len / Type Field    _ 2 bytes
*/
static XStatus SendFrame(XEmac *EmacPtr, Xuint32 FramePayloadSize, Xuint8* InputBuffer,
                         Xuint8 *DestAddress)
{
        Xuint32 Index;
   Xuint8 *FramePtr;
   Xuint8 *AddrPtr = DestAddress;

   FramePtr = (Xuint8 *)TxFrame;

   *FramePtr++ = *AddrPtr++;                  // Writing destination address
   *FramePtr++ = *AddrPtr++;
   *FramePtr++ = *AddrPtr++;
   *FramePtr++ = *AddrPtr++;
   *FramePtr++ = *AddrPtr++;
   *FramePtr++ = *AddrPtr++;
                                                                  // Source address automatically written
                                                                  // just keep place for it
   FramePtr += XEM_MAC_ADDR_SIZE;

                                                                  // Write payload Size
   *(Xuint16 *)FramePtr = (Xuint16)FramePayloadSize;

        FramePtr += 2;
```

```
                        // Write Payload
        for (Index = 0; Index < FramePayloadSize ; Index++)
   {
      *FramePtr++ = *InputBuffer++;
   }

   return XEmac_FifoSend(EmacPtr, (Xuint8 *)TxFrame, FramePayloadSize + XEM_HDR_SIZE);
}



/*
* Pass it a pointer to the message we want to send an ACK for
*
* Destination MAC address for the ACK determined automatically
*
* NOTE: this will send an ACK only for frames that are non ACK Messages
*       The check for this is in the 'if' statement
*
* Returns XST_SUCCESS or XST_FAILURE
*
* XST_FAILURE      if we failed to send
*                          or if we tried to send an ACK for an ACK
*/
XStatus Send_Ack_Message(Xuint8 *MsgBuffer)
{
        Xuint32 mycounter;
   Xuint8 *MessagePtr;
        Xuint8 *SourceAdd;                              // will hold the ack destination address
                                                                // which is the received message source address
```

```
Xuint32 RecvSeqNumber;                          // Received Seq. Num will be sent back in the ACK

Xuint8 *MesgPtr = MsgBuffer;

MesgPtr+=6;                                     // Get destination address Pointer
SourceAdd = MesgPtr;

MesgPtr+=8;

if(*(Xuint16 *)MesgPtr!= ACK_MESSAGE_TYPE)      // Don't send an ACK for an ACK !!
{
        MesgPtr+=2;

        RecvSeqNumber = *(Xuint32 *)MesgPtr;    // Get the sequence number

        // Now Write the Specific Fields Frame To Send in Pre Written ACK Frame

        MessagePtr = (Xuint8 *)AckFrame;  // AckFrame will contain the whole ACK frame

        *MessagePtr++ = *SourceAdd++;           // Copying the Destination Address
        *MessagePtr++ = *SourceAdd++;
        *MessagePtr++ = *SourceAdd++;
        *MessagePtr++ = *SourceAdd++;
        *MessagePtr++ = *SourceAdd++;
        *MessagePtr++ = *SourceAdd++;

        // Get Past Local Address (Automatically Written)   _6 bytes
        // Get Past Ethernet Len Field                                  _2 bytes
        // Get Past Layer 2 Type Field                                          _2 bytes
        // To get to the Sequence Number Position
```

```
                MessagePtr += 10;

                // Writing Sequence Number
                *(Xuint32*)MessagePtr = RecvSeqNumber;

                return XEmac_FifoSend(EmacPtr, (Xuint8 *)AckFrame, ACK_FRAME_SIZE);
        }
        return XST_FAILURE; // we tried to send an ACK for an ACK
}


/*
* Fills the Layer 2 Header and the Payload into TxMessage Buffer
* It calls the SendFrame to fill the Ethernet header and send the frame
*
* Note: the messages sent by this function can only be <= 1490 bytes in length
* returns XST_SUCCESS or XST_FAILURE for just sending the frame
*
* Further reliability should be implemented seperatly
*
*  Message Header Format:    Type                    _ 2 bytes
*                                                    Sequence Number       _ 4 bytes
*                      Message Length     _ 4 bytes
*/
static XStatus SendMessage(Xuint32 MessagePayloadSz, Xuint8* InputBuffer,
                Xuint8 *DestAddress,  Xuint16 TypeofMessage)
{

        Xuint32 CommVarsIndex;
        Xuint32 mycounter;
```

```
Xuint8 *TemporaryPtr;
Xuint8 *MessagePtr;
	Xuint32 FrPaySz;
	Xuint8*TempAddressPtr = DestAddress;

	CommVarsIndex = GetIndex((Xuint8 *)TempAddressPtr);



	TemporaryPtr = TemporaryBuffer;
MessagePtr = (Xuint8 *)TxMessage;

	// Type of Message
	*(Xuint16*)MessagePtr = TypeofMessage;

	MessagePtr+=2;

	// Sequence Number
	*(Xuint32*)TemporaryPtr = CommVars[CommVarsIndex].SeqNum;


	*(Xuint16*)MessagePtr = *(Xuint16 *)TemporaryPtr;

	MessagePtr+=2;
	TemporaryPtr+=2;

	*(Xuint16*)MessagePtr = *(Xuint16 *)TemporaryPtr;

	MessagePtr+=2;
	TemporaryPtr+=2;
```

```
TemporaryPtr = TemporaryBuffer;
*(Xuint32*)TemporaryPtr = MessagePayloadSz;   // Length of Message

*(Xuint16*)MessagePtr = *(Xuint16 *)TemporaryPtr;
MessagePtr+=2;
TemporaryPtr+=2;

*(Xuint16*)MessagePtr = *(Xuint16 *)TemporaryPtr;
MessagePtr+=2;
TemporaryPtr+=2;

TemporaryPtr = InputBuffer;

if(MessagePayloadSz <=MAX_L2_PAYLOAD)
        FrPaySz = MessagePayloadSz + L2_HEADER_SIZE;        // The Frame contains the Message header (10 bytes)
                                                                                                        // and the whole message itself as its payload
  // case of fragmented message
else
{
        if(TypeofMessage == FIRST_FRAGMENT_TYPE || TypeofMessage == INTERNAL_FRAGMENT_TYPE) // frame is full
                FrPaySz = L2_HEADER_SIZE + MAX_L2_PAYLOAD;  // 1490 + 10
        else if (TypeofMessage == LAST_FRAGMENT_TYPE)     // last fragment of a message
        {
                if(MessagePayloadSz % MAX_L2_PAYLOAD == 0)                  // 1490 payload size of last fragment
                        FrPaySz = L2_HEADER_SIZE + MAX_L2_PAYLOAD;   // 1490 + 10
                else
                        FrPaySz = L2_HEADER_SIZE + (MessagePayloadSz % MAX_L2_PAYLOAD);
        }
}
```

```
	for(mycounter = 0 ; mycounter < ( FrPaySz - L2_HEADER_SIZE ) ; mycounter++)
	{
		*MessagePtr++ = *TemporaryPtr++;  // fill layer 2 payload
	}

	return SendFrame(EmacPtr, FrPaySz,  TxMessage, DestAddress);
}


/*
* Uses send message to send the payload passed through the pointer InputBuffer
*
* tried to send it 10 times after which it returns XST_FAILURE
* returns XST_SUCCESS if an ACK is received for this message
*
* Uses different waiting times for resetting the timer.
* Once the timer expires it sets Timer_Expired = 1
*
* Ack_Received is set to 1 by the receive handler function
*
*/
static XStatus SendReliably(Xuint32 MessagePayloadSz, Xuint8 *InputBuffer, Xuint8 *DestAddress, Xuint16 TypeofMessage)
{
	Xuint32 CommVarsIndex;
	Xuint8* TempAddressPtr;
	Xuint8 *MssPtr;
	Xuint8 RetransmissionCounter=0;     // we still did not make retransmissions

	TempAddressPtr = DestAddress;
```

```
	CommVarsIndex = GetIndex((Xuint8 *)TempAddressPtr);


	MssPtr = InputBuffer;
	CommVars[CommVarsIndex].Ack_Received = 0;
	Timer_Expired = 1;                                  // To enter the first time

	while(CommVars[CommVarsIndex].Ack_Received == 0 && RetransmissionCounter < MAX_RETRANSMIT_ATTEMPTS)
	{
		if(Timer_Expired == 1) // The timer wrapped back or we are in for the first time
		{
			RetransmissionCounter++;
			Timer_Expired = 0;
			SendMessage(MessagePayloadSz, MssPtr, DestAddress, TypeofMessage);

			XTmrCtr_SetResetValue(&TimerCounter,TIMER_COUNTER_0,
RstVal[RetransmissionCounter%MAX_RETRANSMIT_ATTEMPTS]);
					// Setting the reset value
			XTmrCtr_Start(&TimerCounter, TIMER_COUNTER_0);
					// Starting to count
			XIntc_Acknowledge(&InterruptController, INTC_INPUT_TIMER);
					// deleting to 'disable' all interrupts previously done by the running counter
			XIntc_Enable(&InterruptController, INTC_INPUT_TIMER);
					// Now we accept Acknowledgments and Pass them to Microblaze
		}
	}
	XIntc_Disable(&InterruptController, INTC_INPUT_TIMER);
					// Don't let the interrupts constantly generated by the counter call the timer handler
	if(RetransmissionCounter == MAX_RETRANSMIT_ATTEMPTS && CommVars[CommVarsIndex].Ack_Received == 0)
		return XST_FAILURE;
```

```
	else
	{
		CommVars[CommVarsIndex].SeqNum = ( CommVars[CommVarsIndex].SeqNum + 1 ) % 0xFFFFFFFF;
		// in case we are not able to transmit,
		// the sequence number stays the same

		return XST_SUCCESS;
	}
}


/*
* Fragments the content pointed to by input buffer and sends the fragmented message across successive
* maximum payload 1490 bytes frames.
*
* In case one of the messages failed to arrive reliably to the receiver,
* the sequence numbers are reset so what they were before starting to send the message
*
* a very unlikely deadlock case is if the last ack that should be received by this sender failed to reach
* us 10 times. In this case, the sender would reset its sequence numbers, whereas the receiver would have the waiting sequence numbers
* after the last received message.
*
* Future Work: Deal properly with this case. Maybe send some special type of message to have like a sequence number handshake
* or agreement.
*/
static XStatus Fragment_and_Send_Reliably(Xuint32 MessagePayloadSz, Xuint8 *InputBuffer, Xuint8 *DestAddress)
{
	Xuint32 CommVarsIndex;
	Xuint32 RememberSeqNum;			// Remembers the sequence number before starting to send for recovery in case of errors
	Xuint8 *CurrentPtr;
```

```
	Xuint32 NumberofFragmentsSent;	// Number of Fragments
	Xuint32 TempSize;			// temporary remaining payload size
	XStatus Worked;				// to know if we could send the message
	Xuint8 *TempAddressPtr;

	TempAddressPtr = DestAddress;
	CommVarsIndex = GetIndex((Xuint8 *)TempAddressPtr);


	RememberSeqNum = CommVars[CommVarsIndex].SeqNum;
	TempSize = MessagePayloadSz;	// At beginning we have the whole message to send
	NumberofFragmentsSent = 0;
	CurrentPtr = InputBuffer;

	if(MessagePayloadSz <= MAX_L2_PAYLOAD)	// can fit in one frame
	{
		Worked = SendReliably(MessagePayloadSz, CurrentPtr, DestAddress, NON_FRAGMENTED_MESSAGE_TYPE);		//
this is an internal frame
		if(Worked == XST_FAILURE)
		{
			CommVars[CommVarsIndex].SeqNum = RememberSeqNum;	// Redundant Here
			xil_printf("Failed to Send the Fragmented Message !!\n\r");
			return XST_FAILURE;					// exit the function
		}
	}


	else if(MessagePayloadSz <= (3 * MAX_L2_PAYLOAD)) // 3 x 1490 bytes where 1490
							// is the maximum message payload size
	{
```

```
while(TempSize != 0)
{
	if(TempSize == MessagePayloadSz)		// this is the first frame (2)
	{
		Worked = SendReliably(MessagePayloadSz, CurrentPtr, DestAddress, FIRST_FRAGMENT_TYPE);
			if(Worked == XST_FAILURE)
			{
				CommVars[CommVarsIndex].SeqNum = RememberSeqNum;
				xil_printf("Failed to Send the Fragmented Message !!\n\r");
				return XST_FAILURE;				// exit the function
			}
		TempSize		-=	MAX_L2_PAYLOAD;		// we have 1490 less bytes to send
		CurrentPtr		+=	MAX_L2_PAYLOAD;		// Increment the Current Pointer by 1490 bytes

	}
	else if(TempSize > MAX_L2_PAYLOAD)		// The Layer 2 Header is 10 bytes and
							// the maximum Ethernet payload is 1500 bytes
	{
		Worked = SendReliably(MessagePayloadSz, CurrentPtr, DestAddress, INTERNAL_FRAGMENT_TYPE); // this is an
internal frame (3)
			if(Worked == XST_FAILURE)
			{
				CommVars[CommVarsIndex].SeqNum = RememberSeqNum;
				xil_printf("Failed to Send the Fragmented Message !!\n\r");
				return XST_FAILURE;			// exit the function
			}
		TempSize		-=	MAX_L2_PAYLOAD; // we have 1490 less bytes to send
		CurrentPtr		+=	MAX_L2_PAYLOAD; // Increment the Current Pointer by 1490 bytes

	}
```

```
                else if(TempSize <= MAX_L2_PAYLOAD)          // Last Frame (4)
                {
                        Worked = SendReliably(MessagePayloadSz, CurrentPtr, DestAddress, LAST_FRAGMENT_TYPE); // this is the last frame

                        if(Worked == XST_FAILURE)
                        {
                                CommVars[CommVarsIndex].SeqNum = RememberSeqNum;
                                xil_printf("Failed to Send the Fragmented Message !!\n\r");
                                return XST_FAILURE;                                  // exit the function
                        }
                        TempSize = 0;
                }
                NumberofFragmentsSent ++;
        }

        return XST_SUCCESS;
    }

    else
    {
        xil_printf("Invalid Length __ Maximum Message Size = 4470 bytes \n\r");

        return XST_FAILURE;
    }

    return XST_SUCCESS;
}
```

SingleMB_HandlerFunctions.h

```
/*****************************************************************************
*
*       Simple Reliable Communication System over Ethernet _ Single MicroBlaze System
*
*       Definitions of the handler functions
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher-John Comis (University of Toronto)
*       at the following email address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/

#ifndef SINGLEMB_HANDLERFUNCTIONS_H
#define SINGLEMB_HANDLERFUNCTIONS_H

#include "SingleMB_GlobalVariables.h"

#include "SingleMB_GlobalVariables.h"
#include "SingleMB_DebuggingFunctions.h"
#include "SingleMB_SendFunctions.h"
```

```
/*
* This function takes care of the incoming messages and writes payload into
* the receive buffers defined in GlobalVariables.h
*
* It also calls the send functions to send Acknowledge frames when
* necessary
*/
static void FifoRecvHandler(void *CallBackRef);


/*
* Sets Timer_Expired to 1 when timer counter wraps around
*
* This value is used in the SendReliably function as an indicator
* that we did not receive an ACK in the acceptable delay
*
* It is disabled not to generate new interrupts.
* We enable it only when we start counting after a send.
*
* CallBackRef is a pointer to the XTmrCtr
*/
static void TimerCounterHandler(void *CallBackRef);

/*
* Checks for errors
*/
static void FifoSendHandler(void *CallBackRef);

/*
* Called in case of Errors to reset the system
```

```
*/
static void ErrorHandler(void *CallBackRef, XStatus Code);

#endif
```

SingleMB_HandlerFunctions.c

```
/*****************************************************************************
*
*       Simple Reliable Communication System over Ethernet _ Single MicroBlaze System
*
*       Source Code for Handler Functions
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher-John Comis (University of Toronto)
*       at the following email address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/


#include "SingleMB_HandlerFunctions.h"
```

```
#include "SingleMB_GlobalVariables.h"
#include "SingleMB_DebuggingFunctions.h"
#include "SingleMB_SendFunctions.h"

/*
* This function takes care of the incoming messages and writes payload into
* the receive buffers defined in GlobalVariables.h
*
* It also calls the send functions to send Acknowledge frames when
* necessary
*/
static void FifoRecvHandler(void *CallBackRef)
{
	Xuint32 CommVarsIndex;
	XStatus Result;
	Xuint32 FrameLen;
	Xuint32 writingcounter;
	Xuint8* TempAddressPtr;
	Xuint8 * WritingPtr;
	Xuint8 * ReadingPtr;

	XIntc_Disable(&InterruptController, INTC_INPUT_EMAC);

	FrameLen = XEM_MAX_FRAME_SIZE;
	Result = XEmac_FifoRecv((XEmac *)CallBackRef, RecvBuffer, &FrameLen);

	if (Result != XST_SUCCESS)                                    // error in reception
	{
		xil_printf("Fifo Receive Error\n\r");
```

```
		XIntc_Enable(&InterruptController, INTC_INPUT_EMAC);
		return;
	}
	else                                                                    // valid message or Ack
	{
		TempAddressPtr = (Xuint8 *)RecvBuffer + XEM_MAC_ADDR_SIZE; // get the source address pointer
		CommVarsIndex = GetIndex((Xuint8 *)TempAddressPtr);

		if(GetType((Xuint8 *)RecvBuffer) == ACK_MESSAGE_TYPE  &&
			GetSeqNum((Xuint8 *)RecvBuffer) == CommVars[CommVarsIndex].SeqNum)
                                                                                // Received an ACK for previous message sent
		{
				XTmrCtr_Stop(&TimerCounter, TIMER_COUNTER_0);
				XIntc_Disable(&InterruptController, INTC_INPUT_TIMER);

				CommVars[CommVarsIndex].Ack_Received =  1;
		}

		if((GetType((Xuint8 *)RecvBuffer) == NON_FRAGMENTED_MESSAGE_TYPE || GetType((Xuint8 *)RecvBuffer) ==
FIRST_FRAGMENT_TYPE)
			&& CommVars[CommVarsIndex].ExpectMore == 1) // error has happened in the last message transmission
		{

		CommVars[CommVarsIndex].WaitingFrameNumber = CommVars[CommVarsIndex].RememberSeqNumReceiver;
                                                                                // Recover from one lost frame in Fragmented
message
                                                                                // whole message is discarded
                                                                                // sequence number is recovered as it was before
reception
```

```
                                                                                    // of first fragment

/*****************************************************************************************
*  User Code : fluch the user's buffers filled with previous fragments of same message
*****************************************************************************************/

/*example*/
xil_printf("Flushing my buffers !!\n\r");

/**********************
*  End of User Code
**********************/



}

if(GetType((Xuint8 *)RecvBuffer) != ACK_MESSAGE_TYPE)
{


        /******* Recovery Algorithm ****/

        //        if we were receiving a fragmented message, in case our last ack was lost,
        //  we have an incremented sequence number whereas the sender does not
        //        so if we get a sequence numbered frame as this remember value, we know that our last ack was lost
        //  so there is a discrepency between our sequence number and that of the sender
        //  to solve this we simply reset our waiting frame number to this value and we receive a new frame

        if(GetSeqNum((Xuint8 *)RecvBuffer) == CommVars[CommVarsIndex].RememberSeqNumReceiver)   // proper incoming number
```

```
{
        CommVars[CommVarsIndex].WaitingFrameNumber = CommVars[CommVarsIndex].RememberSeqNumReceiver;
        xil_printf("Recovering  !!");
}

/*******************************/


if(GetSeqNum((Xuint8 *)RecvBuffer) == CommVars[CommVarsIndex].WaitingFrameNumber) // proper incoming number
{
        Send_Ack_Message(RecvBuffer);                                   // Send an ACK

        if(GetType((Xuint8 *)RecvBuffer) == FIRST_FRAGMENT_TYPE)
        {
                CommVars[CommVarsIndex].ExpectMore = 1;
                CommVars[CommVarsIndex].NbFragRecv = 1;
                // we are expecting more frames

                CommVars[CommVarsIndex].FragLength = MAX_L2_PAYLOAD; // 1490 bytes

                CommVars[CommVarsIndex].RememberSeqNumReceiver = CommVars[CommVarsIndex].WaitingFrameNumber;

                // for recovery in case one of the frames of the message failed

        /* Writing the frame payload into RecvPayload buffer that can hold it all */

                WritingPtr = (Xuint8 *)RecvPayload;                     // Location where to write payload
                ReadingPtr = (Xuint8 *)RecvBuffer;              // location where to read payload
```

```
                        ReadingPtr += 24;                                       // getting to message payload from fragment
frame
                        CommVars[CommVarsIndex].MsgLength =   GetSize(&RecvBuffer);

                        for(writingcounter = 0 ; writingcounter < MAX_L2_PAYLOAD ; writingcounter++)
                        {
                                *WritingPtr++ = *ReadingPtr++;
                        }

                                /*****************************************************************************************
                                *  User Code : Copy the payload from "RecvPayload" Buffer into user buffer

    ******************************************************************************/
                        // Fragment Payload length is in : CommVars[CommVarsIndex].FragLength
                        // Indication of the source address is in : CommVarsIndex
                        // Total Message Payload length is in : CommVars[CommVarsIndex].MsgLength
                        // Number of fragments received (with this one) :     CommVars[CommVarsIndex].NbFragRecv


                        /*Example print statements*/
                                xil_printf("\nType                        : %d\n\r", FIRST_FRAGMENT_TYPE);
                                xil_printf("Index                       : %d\n\r", CommVarsIndex);
                                xil_printf("Fragment Length : %d\n\r", CommVars[CommVarsIndex].FragLength);
                                xil_printf("Message Length    : %d\n\r", CommVars[CommVarsIndex].MsgLength);
                                xil_printf("Nb Frag Received: %d\n\r", CommVars[CommVarsIndex].NbFragRecv);


                                /**********************
                                *  End of User Code
                                CommVars[CommVarsIndex].MsgLength =   GetSize(&RecvBuffer);
```

```
                    **********************/
                }

                CommVars[CommVarsIndex].WaitingFrameNumber = ( CommVars[CommVarsIndex].WaitingFrameNumber + 1 ) %
0xFFFFFFFF;


                if(GetType((Xuint8 *)RecvBuffer) == NON_FRAGMENTED_MESSAGE_TYPE)                    // non
fragmented message encapsulated in one frame
                {
                        CommVars[CommVarsIndex].ExpectMore = 0;
                        CommVars[CommVarsIndex].NbFragRecv = 1;

                        WritingPtr = (Xuint8 *)RecvPayload;                     // Location where to write payload
                        ReadingPtr = (Xuint8 *)RecvBuffer;                  // location where to read payload

                        ReadingPtr += 24;                                               // getting to message payload from fragment
frame
                        CommVars[CommVarsIndex].MsgLength =   GetSize(&RecvBuffer);
                        CommVars[CommVarsIndex].FragLength =   GetSize(&RecvBuffer);


                        for(writingcounter = 0 ; writingcounter <  CommVars[CommVarsIndex].MsgLength ; writingcounter++)
                        {
                                *WritingPtr++ = *ReadingPtr++;
                        }
                                /*****************************************************************************************
                                *  User Code : Copy the payload from "RecvPayload" Buffer into user buffer

        ******************************************************************************/
```

```
                    // Fragment Payload length is in : CommVars[CommVarsIndex].FragLength
                    // Indication of the source address is in : CommVarsIndex
                    // Total Message Payload length is in : CommVars[CommVarsIndex].MsgLength
                    // Number of fragments received (with this one) :     CommVars[CommVarsIndex].NbFragRecv


                    /*Example print statements*/
                        xil_printf("\nType                    : %d\n\r", NON_FRAGMENTED_MESSAGE_TYPE);
                        xil_printf("Index                   : %d\n\r", CommVarsIndex);
                        xil_printf("Fragment Length : %d\n\r", CommVars[CommVarsIndex].FragLength);
                        xil_printf("Message Length    : %d\n\r", CommVars[CommVarsIndex].MsgLength);
                        xil_printf("Nb Frag Received: %d\n\r", CommVars[CommVarsIndex].NbFragRecv);


                        /**********************
                        *  End of User Code
                        **********************/


                }

                if(GetType((Xuint8 *)RecvBuffer) == INTERNAL_FRAGMENT_TYPE)                    // Frame in the middle of
the sequence of frames of a Frag. Messg.
                {
                    CommVars[CommVarsIndex].NbFragRecv ++;
                    CommVars[CommVarsIndex].ExpectMore = 1;

                    ReadingPtr = (Xuint8 *)RecvBuffer;                // location where to read payload

                    ReadingPtr += 24;
```

```
WritingPtr = (Xuint8 *)RecvPayload;                    // Location where to write payload

CommVars[CommVarsIndex].MsgLength =   GetSize(&RecvBuffer);
CommVars[CommVarsIndex].FragLength =   MAX_L2_PAYLOAD;


for(writingcounter = 0 ; writingcounter < MAX_L2_PAYLOAD ; writingcounter++)
{
        *WritingPtr++ = *ReadingPtr++;    // Writing Ptr is still valid from the last time
}

        /*****************************************************************************************
        *  User Code : Copy the payload from "RecvPayload" Buffer into user buffer

*****************************************************************************************/
// Fragment Payload length is in : CommVars[CommVarsIndex].FragLength
// Indication of the source address is in : CommVarsIndex
// Total Message Payload length is in : CommVars[CommVarsIndex].MsgLength
// Number of fragments received (with this one) :     CommVars[CommVarsIndex].NbFragRecv


/*Example print statements*/
        xil_printf("\nType                    : %d\n\r", INTERNAL_FRAGMENT_TYPE);
        xil_printf("Index                   : %d\n\r", CommVarsIndex);
        xil_printf("Fragment Length : %d\n\r", CommVars[CommVarsIndex].FragLength);
        xil_printf("Message Length    : %d\n\r", CommVars[CommVarsIndex].MsgLength);
        xil_printf("Nb Frag Received: %d\n\r", CommVars[CommVarsIndex].NbFragRecv);


        /**********************
```

```
                    *  End of User Code
                    **********************/

                }

                if(GetType((Xuint8 *)RecvBuffer) == LAST_FRAGMENT_TYPE)                    // last frame of a
Fragmented Message
                {
                    CommVars[CommVarsIndex].NbFragRecv++; // we don't need it anymore
                    CommVars[CommVarsIndex].ExpectMore = 0;


                    CommVars[CommVarsIndex].MsgLength = GetSize(&RecvBuffer);

                    if(CommVars[CommVarsIndex].MsgLength % MAX_L2_PAYLOAD == 0)
                        CommVars[CommVarsIndex].FragLength = MAX_L2_PAYLOAD;
                    else
                        CommVars[CommVarsIndex].FragLength = CommVars[CommVarsIndex].MsgLength %
MAX_L2_PAYLOAD;

                    ReadingPtr = (Xuint8 *)RecvBuffer;              // location where to read payload
                    ReadingPtr += 24;
                    WritingPtr = (Xuint8 *)RecvPayload;                    // Location where to write payload

                    for(writingcounter = 0 ; writingcounter < CommVars[CommVarsIndex].FragLength ; writingcounter++)
                    {
                        *WritingPtr++ = *ReadingPtr++;    // Writing Ptr is still valid from the last time
                    }
```

```
                        /*****************************************************************************
                        *  User Code : Copy the payload from "RecvPayload" Buffer into user buffer

*****************************************************************************/
                        // Fragment Payload length is in : CommVars[CommVarsIndex].FragLength
                        // Indication of the source address is in : CommVarsIndex
                        // Total Message Payload length is in : CommVars[CommVarsIndex].MsgLength
                        // Number of fragments received (with this one) :     CommVars[CommVarsIndex].NbFragRecv


                        /*Example print statements*/
                                xil_printf("\nType                    : %d\n\r", LAST_FRAGMENT_TYPE);
                                xil_printf("Index                   : %d\n\r", CommVarsIndex);
                                xil_printf("Fragment Length : %d\n\r", CommVars[CommVarsIndex].FragLength);
                                xil_printf("Message Length    : %d\n\r", CommVars[CommVarsIndex].MsgLength);
                                xil_printf("Nb Frag Received: %d\n\r", CommVars[CommVarsIndex].NbFragRecv);


                                /**********************
                                *  End of User Code
                                **********************/



                        }
                }
                else if(GetSeqNum((Xuint8 *)RecvBuffer) == CommVars[CommVarsIndex].WaitingFrameNumber - 1)
                {
                        Send_Ack_Message(RecvBuffer);
```

```
                                        // Send an ACK Message (our previous ack was probably lost
                        }
                }

        }


//      Print_Frame_Fields(RecvBuffer, FrameLen);
//      Print_Message_Fields(RecvBuffer, FrameLen - 18);

        XIntc_Enable(&InterruptController, INTC_INPUT_EMAC);
}



/*
* Sets Timer_Expired to 1 when timer counter wraps around
*
* This value is used in the SendReliably function as an indicator
* that we did not receive an ACK in the acceptable delay
*
* It is disabled not to generate new interrupts.
* We enable it only when we start counting after a send.
*
* CallBackRef is a pointer to the XTmrCtr
*/
static void TimerCounterHandler(void *CallBackRef)
{
        Timer_Expired = 1;
   xil_printf("ACK Time Exp.\n\r");
```

```
	XIntc_Disable(&InterruptController, INTC_INPUT_TIMER);
}


/*
* Checks for errors
*/
static void FifoSendHandler(void *CallBackRef)
{
   XEmac_GetStats((XEmac *)CallBackRef, &Stats);

	if (Stats.XmitLateCollisionErrors || Stats.XmitExcessDeferral)
   {
		xil_printf("Late Collision Number : %d\n\r", Stats.XmitLateCollisionErrors);
		xil_printf("Late Deferrals Number : %d\n\r", Stats.XmitExcessDeferral);
		xil_printf("Error in FIFO Sending !!\n\r");
		PrintEmacStats(Stats);
   }

	XEmac_ClearStats((XEmac *)CallBackRef);
}


/*
* Called in case of Errors to reset the system
*/
static void ErrorHandler(void *CallBackRef, XStatus Code)
{
	xil_printf("Starting FIFO Error Handler\n\r");
```

```
   if (Code == XST_RESET_ERROR)
   {
      XEmac_Reset((XEmac *)CallBackRef);                                    // Reset EMAC

      (void)XEmac_SetMacAddress((XEmac *)CallBackRef, LocalAddress);        // SET MAC ADDRESS
      (void)XEmac_SetOptions((XEmac *)CallBackRef, TEST_1_OPTIONS);         // SET OPTIONS

      (void)XEmac_Start((XEmac *)CallBackRef);                                  // Start EMAC
   }
         xil_printf("Exiting FIFO Error Handler\n\r");
}
```

SingleMB_InitializationFunctions.h

```
/*****************************************************************************
*
*       Simple Reliable Communication System over Ethernet _ Single MicroBlaze System
*
*       Initialization Functions Definitions
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher-John Comis (University of Toronto)
*       at the following email address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/

#ifndef SINGLEMB_INITIALIZATIONFUNCTIONS_H
#define SINGLEMB_INITIALIZATIONFUNCTIONS_H


#include "SingleMB_GlobalVariables.h"
#include "SingleMB_HandlerFunctions.h"

/*
```

```
* Initializes the GPio
* Led is an output
*/
static XStatus Initialize_GPio();


/*
* Initializes the ACK Frame common fields
* so that we don't do it every time we send an ACK
* at run time.
*
* This is to reduce the computational overhead
* The Ack Frame is contained in the Buffer ACKBuffer[60]
*
* ACK Format:
*
*   DST ADDRESS_SRC ADDRESS_ETHERNET LEN/TYPE FIELD_MESSAGE TYPE_SEQUENCE NUMBERS_MESSAGE LEN_PAYLOAD
*    unknown     automatic          known                    known          unknown                  known    known
*/
static void InitializeACKFrameCommonFields();

/*
* Setup Interrupt System
*
* EMAC has highest priority                0
* TIMER COUNTER has lowest priority 1
*/
static XStatus SetupInterruptSystem(XEmac *EmacPtr);

/*
```

```
* Initialize the Timer Counter
*/
static XStatus InitializeTimerCounter();

/*
* Initialize the EMAC Module
*/
static XStatus InitializeEmac();

/*
* This function is called from Initialize Parameters (which is the only one called from main)
* and hence there is no need to
* call it from main.
*
* It initializes the Communication Parameters Structure contents
*
* It assumes we have 5 boards (check the NUMBER_OF_BOARDS variable in Global Variables)
*/
void Initialize_CommVars();

/*
* This function should be called from main
* It calls all the other functions.
*/
void Initialize_Parameters();

#endif
```

SingleMB_InitializationFunctions.c

```
/*****************************************************************************
*
*       Simple Reliable Communication System over Ethernet _ Single MicroBlaze System
*
*       Source Code for Handler Functions
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher-John Comis (University of Toronto)
*       at the following email address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/


#include "SingleMB_InitializationFunctions.h"
#include "SingleMB_GlobalVariables.h"
#include "SingleMB_MACAddresses.h"

/*
* Initializes the GPio
*
```

```
* Led is an output
*/
static XStatus Initialize_GPio()
{
        XStatus Status;
   Status = XGpio_Initialize(&Gpio, XPAR_OPB_GPIO_0_DEVICE_ID);
   if (Status != XST_SUCCESS)
   {
      xil_printf("GPIO initialization error !!\n\r");
   }
        else
                xil_printf("GPIO initialization succeeded !!\n\r");
   /* Set the direction for all signals to be inputs except the LED
    * outputs */
   XGpio_SetDataDirection(&Gpio, ~LED);

        return Status;
}



/*
*
* Initializes the ACK Frame common fields
* so that we don't do it every time we send an ACK
* at run time.
*
* This is to reduce the computational overhead
* The Ack Frame is contained in the Buffer ACKBuffer[60]
*
```

```
* ACK Format:
*
*   DST ADDRESS_SRC ADDRESS_ETHERNET LEN/TYPE FIELD_MESSAGE TYPE_SEQUENCE NUMBERS_MESSAGE LEN_PAYLOAD
*    unknown     automatic          known                    known          unknown                  known    known
*/
static void InitializeACKFrameCommonFields()
{
        Xuint32 loopcounter;
        Xuint8* ACKBuffPointer;
        ACKBuffPointer = (Xuint8 *) AckFrame;

                // Get Passed Destination and Source MAC Address
        ACKBuffPointer += (2 * XEM_MAC_ADDR_SIZE);

                // Write Known ETHERNET LEN / TYPE Field _ 46
        *(Xuint16 *)ACKBuffPointer = ACK_L1_PAYLOAD_SIZE;

                // Advance the pointer to the position of Layer 2 Header
        ACKBuffPointer += 2;

                // Write Known Message Type _ 1
        *(Xuint16 *)ACKBuffPointer = ACK_MESSAGE_TYPE;

                // Get Passed SeqNum Field _ Get to Layer2 Len Field
        ACKBuffPointer += 6;

                // Write Known Layer 2 Payload Length Field
        *(Xuint32 *)ACKBuffPointer = ACK_L2_PAYLOAD_SIZE;

                // Advance Pointer to the Layer2 Known ACK Payload
```

```
		ACKBuffPointer += 4;

			//Writing Known Layer2 ACK Payload
		for(loopcounter = 0 ; loopcounter < ACK_L2_PAYLOAD_SIZE ; loopcounter++)
		{
			*(Xuint8 *)ACKBuffPointer++ = 0xFF;
		}
}


/*
* Setup Interrupt System
*
* EMAC has highest priority			0
* TIMER COUNTER has lowest priority 1
*/
static XStatus SetupInterruptSystem(XEmac *EmacPtr)			/* Setup Interrupt System */
{
   XStatus Result;

   microblaze_enable_interrupts();

   Result = XIntc_Initialize(&InterruptController, INTC_DEVICE_ID); // 0 is DEVICE ID for the Interrupt Controller
        if (Result != XST_SUCCESS)
   {
      xil_printf("Error in Xintc Initialization !!\n\r");
			return Result;
   }
        else
      xil_printf("Xintc Initialization Succeeded !!\n\r");
```

```
	Result = XIntc_Connect(&InterruptController, INTC_INPUT_TIMER, (XInterruptHandler)XTmrCtr_InterruptHandler, &TimerCounter);
if (Result != XST_SUCCESS)
{
   xil_printf("Error in Xintc Connect to Timer !!\n\r");
		return Result;
}
	else
   xil_printf("Xintc Connect Succeeded to Timer!!\n\r");


	Result = XIntc_Connect(&InterruptController, INTC_INPUT_EMAC, (XInterruptHandler)XEmac_IntrHandlerFifo, EmacPtr);
if (Result != XST_SUCCESS)
{
		return Result;
   xil_printf("Error in Xintc Connect to EMAC!!\n\r");
}
	else
   xil_printf("Xintc Connect Succeeded to EMAC!!\n\r");
                                                        // all interrupts will be serviced in order of occurence
	Result = XIntc_SetOptions(&InterruptController, XIN_SVC_ALL_ISRS_OPTION);
	if (Result != XST_SUCCESS)
{
		return Result;
   xil_printf("Error in Xintc SetOptions !!\n\r");
}
	else
   xil_printf("Xintc SetOptions Succeeded !!\n\r");
```

```
        Result = XIntc_Start(&InterruptController, XIN_REAL_MODE);
        if (Result != XST_SUCCESS)
   {
                return Result;
      xil_printf("Error in Xintc Start !!\n\r");
   }
        else
      xil_printf("Xintc Start Succeeded !!\n\r");


   XIntc_Enable(&InterruptController, INTC_INPUT_EMAC); // enable the EMAC interrupt
        //INTC_INPUT_TIMER is enabled in the SendReliably function after a send

   return XST_SUCCESS;
}

/*
* Initialize the Timer Counter
*/
static XStatus InitializeTimerCounter()
{
        XStatus Status;
                // Disabling the counter if some are running with low level funtions
        XTmrCtr_mDisable(0x80010400, TIMER_COUNTER_0);

                // Disable the input of the interrupt controller
        XIntc_mMasterDisable(0x80010500);
```

```
    /************************** Timer Setup **********************************/

    Status = XTmrCtr_Initialize(&TimerCounter, TIMER_COUNTER_0);

    if (Status != XST_SUCCESS)
    {
       xil_printf("Timer counter initialization error\n\r\r");
                    return Status;
    }

    Status = XTmrCtr_SelfTest(&TimerCounter, TIMER_COUNTER_0);
    if (Status != XST_SUCCESS)
    {
       xil_printf("Timer counter self-test error\n\r");
                    return Status;
    }

    XTmrCtr_SetHandler(&TimerCounter, TimerCounterHandler, &TimerCounter);

    XTmrCtr_SetOptions(&TimerCounter, TIMER_COUNTER_0, XTC_INT_MODE_OPTION | XTC_AUTO_RELOAD_OPTION);

    XTmrCtr_SetResetValue(&TimerCounter, TIMER_COUNTER_0, 0xF0000000);

    return XST_SUCCESS;
}

/*
* Initialize the EMAC Module
*/
static XStatus InitializeEmac()
```

```
{
        XEmac_Config *ConfigPtr;
        XStatus Result;
        Xuint32 DeviceId;
        DeviceId = 0;
        EmacPtr = & Emac;

   ConfigPtr = XEmac_LookupConfig(DeviceId);
   ConfigPtr->IpIfDmaConfig = XEM_CFG_NO_DMA;

   Result = XEmac_Initialize(EmacPtr, DeviceId);
   if (Result != XST_SUCCESS)
   {
                xil_printf("Error in EMAC Initialization !!\n\r");
      return Result;
   }
        else
                xil_printf("Initialization Succeeded !!\n\r");


   Result = XEmac_SelfTest(EmacPtr);
   if (Result != XST_SUCCESS)
   {
                xil_printf("Self Test Error !!\n\r");
      return Result;
   }
        else
                xil_printf("Self Test Succeeded !!\n\r");

   Result = XEmac_SetOptions(EmacPtr, TEST_1_OPTIONS);
```

```
if (Result != XST_SUCCESS)
{
		xil_printf("Set Options Error !!\n\r");
   return Result;
}
	else
		xil_printf("Set Options Succeeded !!\n\r");

Result = XEmac_SetMacAddress(EmacPtr, LocalAddress);
if (Result != XST_SUCCESS)
{
		xil_printf("Set MAC Address Error !!\n\r");
   return Result;
}
	else
		xil_printf("Set MAC Address Succeeded !!\n\r");

XEmac_SetFifoSendHandler(EmacPtr, EmacPtr, FifoSendHandler);
XEmac_SetFifoRecvHandler(EmacPtr, EmacPtr, FifoRecvHandler);
XEmac_SetErrorHandler(EmacPtr, EmacPtr, ErrorHandler);


	XEmac_ClearStats(EmacPtr);
Result = XEmac_Start(EmacPtr);
if (Result != XST_SUCCESS)
{
		xil_printf("EMAC Start Error !!\n\r");
   return Result;
}
	else
```

```
                xil_printf("EMAC Start Success !!\n\r");

        return XST_SUCCESS;
}



/*
* This function is called from Initialize Parameters (which is the only one called from main)
* and hence there is no need to
* call it from main.
*
* It initializes the Communication Parameters Structure contents
*
* It assumes we have 5 boards (check the NUMBER_OF_BOARDS variable in Global Variables)
*/
void Initialize_CommVars()
{
        Xuint8 cntr;

        for(cntr=0 ; cntr < NUMBER_OF_BOARDS ; cntr++)
        {
                CommVars[cntr].SeqNum = 0;
                CommVars[cntr].WaitingFrameNumber = 0;
                CommVars[cntr].Ack_Received = 0;
                CommVars[cntr].ExpectMore = 0;
                CommVars[cntr].RememberSeqNumReceiver = 0;
                CommVars[cntr].NbFragRecv = 0;
                CommVars[cntr].MsgLength = 0;
```

```
                CommVars[cntr].FragLength = 0;
        }
}


/*
* This function should be called from main
* It calls all the other functions.
*/
void Initialize_Parameters()
{
        XStatus Result;

        Timer_Expired = 1;
        reload_flag=0;

        Initialize_CommVars();

        InitializeACKFrameCommonFields();

        Result = InitializeEmac();
        if (Result != XST_SUCCESS)
   {
                xil_printf("EMAC Initialization Error !!\n\r");
      return;
   }
        else
                xil_printf("EMAC Initialization Success !!\n\r");
```

```
        Result = InitializeTimerCounter();
        if (Result != XST_SUCCESS)
   {
                xil_printf("TimerCounter Initialization Error !!\n\r");
      return;
   }
        else
                xil_printf("TimerCounter Initialization Success !!\n\r");

   Result = SetupInterruptSystem(EmacPtr);
   if (Result != XST_SUCCESS)
   {
                xil_printf("Set Interrupt System Error !!\n\r");
      return;
   }
        else
                xil_printf("Set Interrupt System Success !!\n\r");


        Result = Initialize_GPio();
   if (Result != XST_SUCCESS)
   {
                xil_printf("GPio Initialization Error !!\n\r");
      return;
   }
        else
                xil_printf("GPio Initialization Success !!\n\r");
}
```

SingleMB_MACAddresses.h

```
/*****************************************************************************
*
*       Simple Reliable Communication System over Ethernet _ Single MicroBlaze System
*
*       MAC Addresses definitions and selection of the local address
*       Definition for some MAC Address operation functions
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher-John Comis (University of Toronto)
*       at the following email address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/

#ifndef SINGLEMB_MACADDRESSES_H
#defineSINGLEMB_MACADDRESSES_H


/*************************  Number of Boards  ****************************************/


#define NUMBER_OF_BOARDS                5
```

```
/************************** MAC Addresses Definitions ********************************/

static Xuint8 Address0[XEM_MAC_ADDR_SIZE] =         //  48 bits  address
{
   0x06, 0x05, 0x04, 0x03, 0x02, 0x00
};


static Xuint8 Address1[XEM_MAC_ADDR_SIZE] =                 //  48 bits  address
{
   0x06, 0x05, 0x04, 0x03, 0x02, 0x01
};


static Xuint8 Address2[XEM_MAC_ADDR_SIZE] =         //  48 bits  address
{
   0x06, 0x05, 0x04, 0x03, 0x02, 0x02
};

static Xuint8 Address3[XEM_MAC_ADDR_SIZE] =         //  48 bits  address
{
   0x06, 0x05, 0x04, 0x03, 0x02, 0x03
};

static Xuint8 Address4[XEM_MAC_ADDR_SIZE] =         //  48 bits  address
{
   0x06, 0x05, 0x04, 0x03, 0x02, 0x04
};
```

```
/************************  Local MAC Address Selection   ******************************/

Xuint8 * LocalAddress = Address1;

/************************  Function Definitions   *************************************/

/*
* This function takes a pointer to two MAC Addresses and
* compares them
* returns XST_SUCCESS if they match
*/
XStatus CompareMACAddresses(Xuint8 *MAC1, Xuint8 *MAC2);

/*
* This function returns the appropriate index for the CommVars Array
* according to the MACAddress Field passed to it
* returns -1 for FAILURE
*/
Xuint32 GetIndex(Xuint8 *MACAddress);




#endif
```

SingleMB_MACAddresses.c

```
/*****************************************************************************
*
*       Simple Reliable Communication System over Ethernet _ Single MicroBlaze System
*
*       MAC Addresses definitions and selection of the local address
*       Definition for some MAC Address operation functions
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher-John Comis (University of Toronto)
*       at the following email address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/

#include "SingleMB_MACAddresses.h"

/*
* This function takes a pointer to two MAC Addresses and
* compares them
* returns XST_SUCCESS if they match
```

```
*/
XStatus CompareMACAddresses(Xuint8 *MAC1, Xuint8 *MAC2)
{
	Xuint8 cnt;
	Xuint8 *ptr1 = MAC1;
	Xuint8 *ptr2 = MAC2;

	for(cnt = 0 ; cnt < XEM_MAC_ADDR_SIZE ; cnt++)
	{
		if( *(Xuint8 *)ptr1  !=   *(Xuint8 *)ptr2 )
			return XST_FAILURE;

		ptr1 ++;
		ptr2 ++;

	}
	return XST_SUCCESS;
}

/*
* This function returns the appropriate index for the CommVars Array
* according to the MACAddress Field passed to it
* returns -1 for FAILURE
*/
Xuint32 GetIndex(Xuint8 *MACAddress)
{
	if(CompareMACAddresses((Xuint8*)Address0, (Xuint8 *)MACAddress) == XST_SUCCESS )
	{
		return 0 ;
	}
```

```
        else if(CompareMACAddresses((Xuint8*)Address1, (Xuint8 *)MACAddress) == XST_SUCCESS )
        {
                return 1 ;
        }
        else if(CompareMACAddresses((Xuint8*)Address2, (Xuint8 *)MACAddress) == XST_SUCCESS )
        {
                return 2 ;
        }
        else if(CompareMACAddresses((Xuint8*)Address3, (Xuint8 *)MACAddress) == XST_SUCCESS )
        {
                return 3 ;
        }
        else if(CompareMACAddresses((Xuint8*)Address4, (Xuint8 *)MACAddress) == XST_SUCCESS )
        {
                return 4 ;
        }
        else
                return -1;   // for errors

}
```

System 2: Dual MicroBlaze Communication System

This system consists of two MicroBlaze system running on each multimedia board and communicating via the FSLs. MicroBlaze_0 deals with the networking protocol, whereas MicroBlaze_1 provides the user with an interface where he can use high level functions without caring about the complication of the design. This separation of the two MicroBlazes also allows the code of each MicroBlaze to run in parallel with the code of the second MicroBlaze. MicroBlaze_0 and MicroBlaze_1 communicate using the FSLs. Since the whole system cannot contain more than 64 KB of memory on chip, and because we have two MicroBlazes on each FPGA, each one is restricted to have a maximum of 32 KB of memory on chip. However, this is not enough for the source code that has to run on MicroBlaze_0, and that's why, an external memory (ZBTs) was used for MicroBlaze_0, whereas the code and variables of MicroBlaze_1 run on local 32 KB memory.

Hardware Configuration

This system consists of two MicroBlaze Modules. Since the source code of **microblaze_0** runs from external ZBT Memories, three modules were added. These are the **opb_emc**, **dcm_module** and the **clk_align** module. These modules are added to interconnect and synchronize **microblaze_0** to ZBTs.

Two timers are added and are used by **microblaze_0**, one of them is for retransmission purpose while the second one is simply to provide an interrupt capability from **microblaze_0** to **microblaze_1**. Also each MicroBlaze uses an interrupt controller.

Peripheral	HW Ver	Instance	Base Address	High Address	Min
microblaze	2.10.a	microblaze_0			
opb_mdm	2.00.a	debug_module	0x00016000	0x000160ff	0x100
lmb_bram_if_cntlr	1.00.b	dlmb_cntlr	0x00000000	0x00007fff	0x800
lmb_bram_if_cntlr	1.00.b	ilmb_cntlr	0x00000000	0x00007fff	0x800
bram_block	1.00.a	lmb_bram			
opb_uartlite	1.00.b	RS232	0x00016100	0x000161ff	0x100
microblaze	2.10.a	microblaze_1			
bram_block	1.00.a	lmb_bram_1			
lmb_bram_if_cntlr	1.00.b	dlmb_cntlr_1	0x00008000	0x0000ffff	0x800
lmb_bram_if_cntlr	1.00.b	ilmb_cntlr_1	0x00008000	0x0000ffff	0x800
opb_gpio	3.01.a	opb_gpio_0	0x80010200	0x800103ff	0x1FF
opb_timer	1.00.b	opb_timer_0	0x80010400	0x800104ff	0x100
opb_intc	1.00.c	opb_intc_0	0x80010500	0x800105ff	0x20
opb_ethernet	1.00.m	opb_ethernet_0	0x80020000	0x80023fff	0x4000
opb_intc	1.00.c	opb_intc_1	0x80010600	0x800106ff	0x20
opb_uartlite	1.00.b	RS232_1	0x00016200	0x000162ff	0x100
opb_timer	1.00.b	opb_timer_1	0x80010700	0x800107ff	0x100
opb_emc	1.10.b	ZBT_512Kx32	0x80400400	0x80400401	0x02
dcm_module	1.00.a	dcm_module_0			
clk_align	1.00.a	clk_align_0			

- MicroBlaze_0 and MicroBlaze_1: The processors in the system. MicroBlaze_0 deals with all the communication protocols and is connected to MicroBlaze_1 via two FSLs, one of which acts as a slave, the other one as a master. MicroBlaze_1 will be used by the user of the system who will not have to worry about MicroBlaze_0.

- FSL_0 and FSL_1: the fast simplex links used to perform data transfer between MicroBlaze_0 and MicroBlaze_1. Note that because the operations used for data transfers on the FSLs are blocking reads and writes, a simple FSL communication protocol was created to suit the whole system architecture.

- MB_OPB_0 and MB_OPB_1: On Chip Peripheral Busses are used to interconnect the two processors to their respective peripheral devices used.

- OPB GPIO: peripheral general purpose input output device connected to the MicroBlaze systems using the OPB, used to take input from a DIP Switch and drive output to a user LED.

- OPB MDM: peripheral Microprocessor Debug Module connected to the MicroBlaze_0 using the OPB.

- OPB RS232 and OPB RS232_1: peripheral serial link devices connected respectively to MicroBlaze_0 and MicroBlaze_1 using the OPB and used to drive the output serial link that is used to print statements for debugging and verification on a HyperTerminal on a computer. Note that only one serial link can be used at a time, and in the default system, RS232_1 is physically connected to the serial link controller. This can be modified from the hardware settings windows in XPS.

- OPB ETHERNET: peripheral Ethernet controller device connected to MicroBlaze_0 using the OPB.

- OPB TIMER and OPB TIMER_1: peripheral timer counter devices connected to the MicroBlaze_0 using the OPB. TIMER is used to notify the sender that it has to retransmit a frame or abort, whereas TIMER_1 is used to generate an interrupt used as the input of the interrupt controller INTC of MicroBlaze_1 to notify it that a received frame payload is ready to be transferred via the FSLs.

- OPB INTC and OPB INTC_1: peripheral interrupt controller devices connected respectively to MicroBlaze_0 and MicroBlaze_1 using the OPB, and with the processor interrupt signal directly connected to the processor interrupt signal of the MicroBlaze. INTC takes as an input the interrupt output of the Ethernet controller (high priority) and the interrupt output of the timer counter TIMER(low priority). INTC_1 takes as an input the output of the interrupt signal of TIMER_1.

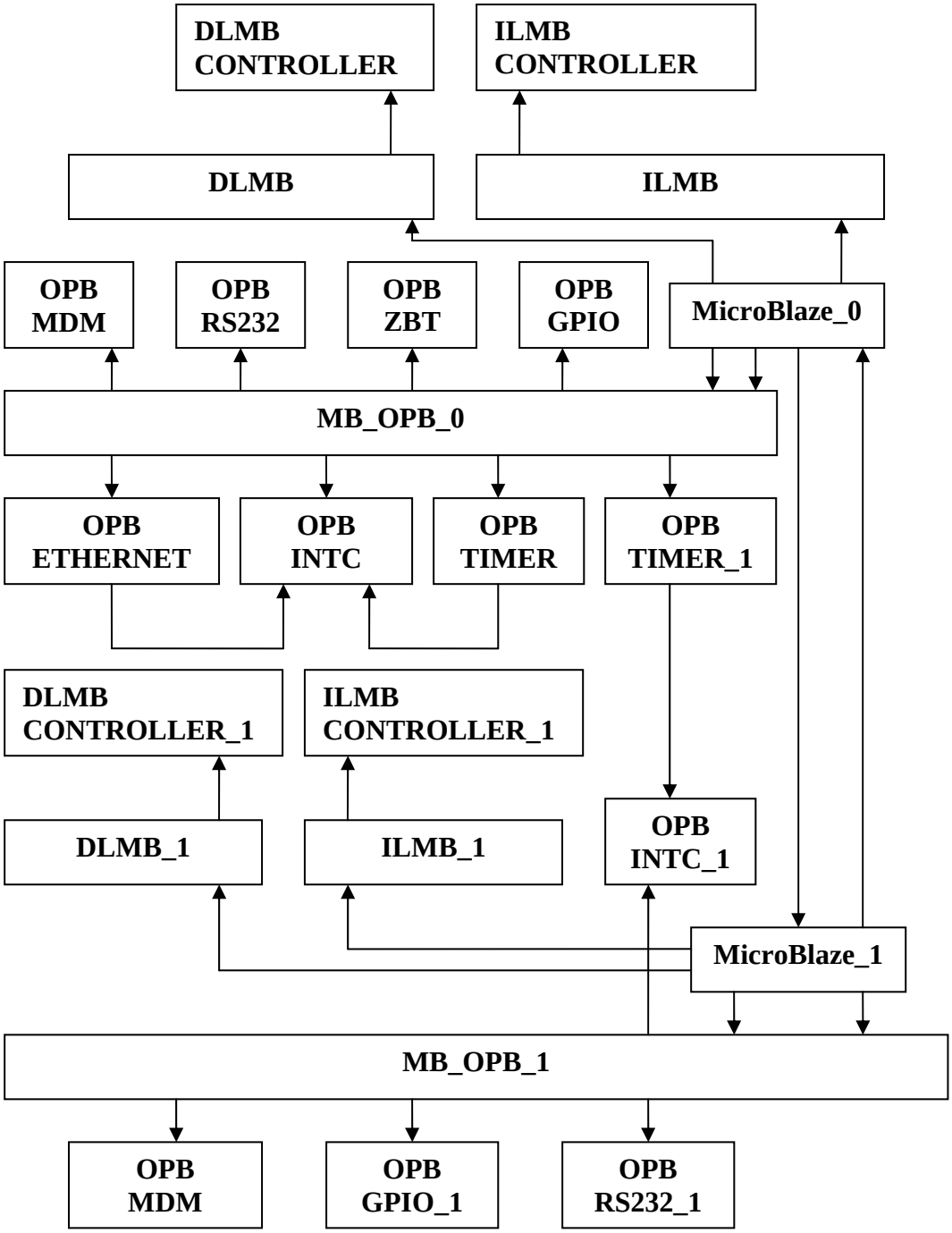
- DLMB and DLMB_1: Data Local Memory Bus connected to the corresponding MicroBlaze.

- ILMB and ILMB_1: Instruction Local Memory Bus connected to the corresponding MicroBlaze.

- DLMB CONTROLLER and DLMB CONTROLLER_1: Data Local Memory Bus Controller connected respectively to DLMB and DLMB_1.

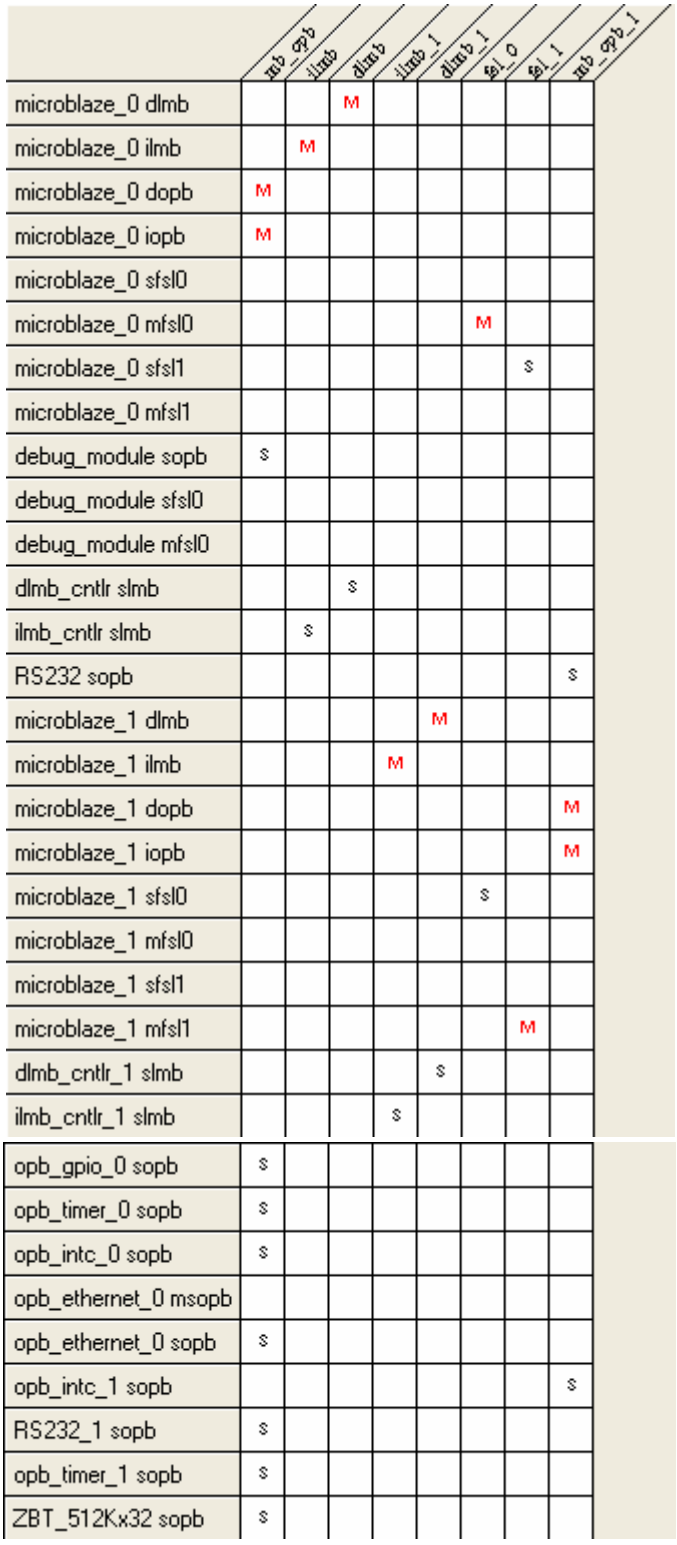
- ILMB CONTROLLER and ILMB CONTROLLER_1: Instruction Local Memory Bus Controller connected respectively to ILMB and ILMB_1.

Here is a schematic of the hardware modules in the system:



Bus Connections

Note that **M** means master and **s** means slave. FSLs are added to allow data exchange between the two MicroBlazes. Notice how **microblaze_0** is master on **FSL_0** and slave on **FSL_1** and vice versa for **microblaze_1**. Also notice how the ZBTs are connected as slave on the opb bus connected to **microblaze_0**.



	mb_opb	ilmb	dlmb	ilmb_1	dlmb_1	fsl_0	fsl_1	mb_opb_1
microblaze_0 dlmb			M					
microblaze_0 ilmb		M						
microblaze_0 dopb	M							
microblaze_0 iopb	M							
microblaze_0 sfsl0								
microblaze_0 mfsl0						M		
microblaze_0 sfsl1							s	
microblaze_0 mfsl1								
debug_module sopb	s							
debug_module sfsl0								
debug_module mfsl0								
dlmb_cntlr slmb			s					
ilmb_cntlr slmb		s						
RS232 sopb								s
microblaze_1 dlmb					M			
microblaze_1 ilmb				M				
microblaze_1 dopb								M
microblaze_1 iopb								M
microblaze_1 sfsl0						s		
microblaze_1 mfsl0								
microblaze_1 sfsl1								
microblaze_1 mfsl1							M	
dlmb_cntlr_1 slmb					s			
ilmb_cntlr_1 slmb				s				
opb_gpio_0 sopb	s							
opb_timer_0 sopb	s							
opb_intc_0 sopb	s							
opb_ethernet_0 msopb								
opb_ethernet_0 sopb	s							
opb_intc_1 sopb								s
RS232_1 sopb	s							
opb_timer_1 sopb	s							
ZBT_512Kx32 sopb	s							

Ports

The interrupt outputs of **timer_0** and the **opb_ethernet** modules are connected to the input of **intc_0**, the interrupt controller used by **microblaze_0** for the protocol, with the emac module having the highest interrupt priority. The **timer_1** module is used by **microblaze_0** to interrupt **microblaze_1**. Note that the output **timer_intr_1** is connected to the input of **intc_1**, which is connected to **microblaze_1**.

The ports connected with **external scope** are connected to external pins. The description of these connections is in the .ucf file.

Instance	Port Name	Net Name	Polarity	Scope	Range	Class	Sensitivity
system	ZBT_512Kx32_EMC_CLK_FEEDBACK_IN	zbt_dcm_feedback	IN	External			
system	RS232_req_to_send	net_gnd	OUT	External			
system	PHY_slew1	net_vcc	OUT	External			
system	PHY_slew2	net_vcc	OUT	External			
system	sys_rst	sys_rst_s	IN	External			
system	sys_clk	sys_clk_s	IN	External		CLK	
system	ZBT_512Kx32_EMC_CLK_OUT	zbt_dcm_clk	OUT	External			
system	ZBT_512Kx32_EMC_CLK_FEEDBACK_OUT	zbt_dcm_clk	OUT	External			
opb_timer_1	Interrupt	timer_intr_1	O	Internal		INTERRUPT	LEVEL_HIGH
opb_timer_0	Interrupt	timer_intr	O	Internal		INTERRUPT	LEVEL_HIGH
opb_intc_1	Intr	timer_intr_1	I	Internal		INTERRUPT	
opb_intc_1	Irq	IntConnection3	O	Internal		INTERRUPT	
opb_intc_0	Irq	IntConnection2	O	Internal		INTERRUPT	
opb_intc_0	Intr	timer_intr & emac_intr	I	Internal		INTERRUPT	
opb_gpio_0	GPIO_IO	opb_gpio_0_GPIO_IO	IO	External	[0:1]		
opb_ether...	PHY_tx_en	opb_ethernet_0_PHY_tx_en	O	External			
opb_ether...	IP2INTC_Irpt	emac_intr	O	Internal		INTERRUPT	LEVEL_HIGH
opb_ether...	PHY_crs	opb_ethernet_0_PHY_crs	I	External			
opb_ether...	PHY_dv	opb_ethernet_0_PHY_dv	I	External			
opb_ether...	PHY_rx_clk	opb_ethernet_0_PHY_rx_clk	I	External			
opb_ether...	PHY_rx_data	opb_ethernet_0_PHY_rx_data	I	External	[3:0]		
opb_ether...	PHY_rx_er	opb_ethernet_0_PHY_rx_er	I	External			
opb_ether...	PHY_tx_er	opb_ethernet_0_PHY_tx_er	O	External			
opb_ether...	PHY_Mii_clk	opb_ethernet_0_PHY_Mii_clk	IO	External			
opb_ether...	PHY_Mii_data	opb_ethernet_0_PHY_Mii_data	IO	External			
opb_ether...	PHY_col	opb_ethernet_0_PHY_col	I	External			
opb_ether...	PHY_tx_clk	opb_ethernet_0_PHY_tx_clk	I	External			
opb_ether...	PHY_tx_data	opb_ethernet_0_PHY_tx_data	O	External	[3:0]		

microblaze_1	CLK	sys_clk_s	I	Internal		CLK	
microblaze_1	INTERRUPT	IntConnection3	I	Internal		INTERRUPT	LEVEL_HIGH
microblaze_0	DBG_TDO	DBG_TDO_0	O	Internal			
microblaze_0	DBG_CLK	DBG_CLK_0	I	Internal			
microblaze_0	DBG_UPDATE	DBG_UPDATE_0	I	Internal			
microblaze_0	CLK	sys_clk_s	I	Internal		CLK	
microblaze_0	INTERRUPT	IntConnection2	I	Internal		INTERRUPT	LEVEL_HIGH
microblaze_0	DBG_CAPTURE	DBG_CAPTURE_0	I	Internal			
microblaze_0	DBG_TDI	DBG_TDI_0	I	Internal			
microblaze_0	DBG_REG_EN	DBG_REG_EN_0	I	Internal			
mb_opb_1	OPB_Clk	sys_clk_s	I	Internal		CLK	
mb_opb_1	SYS_Rst	sys_rst_out	I	Internal			
mb_opb	OPB_Clk	sys_clk_s	I	Internal		CLK	
mb_opb	SYS_Rst	sys_rst_out	I	Internal			
ilmb_1	SYS_Rst	sys_rst_out	I	Internal			
ilmb_1	LMB_Clk	sys_clk_s	I	Internal		CLK	
ilmb	SYS_Rst	sys_rst_out	I	Internal			
ilmb	LMB_Clk	sys_clk_s	I	Internal		CLK	
fsl_1	FSL_Clk	sys_clk_s	I	Internal		CLK	
fsl_1	SYS_Rst	sys_rst_out	I	Internal			
fsl_0	FSL_Clk	sys_clk_s	I	Internal		CLK	
fsl_0	SYS_Rst	sys_rst_out	I	Internal			
dlmb_1	LMB_Clk	sys_clk_s	I	Internal		CLK	
dlmb_1	SYS_Rst	sys_rst_out	I	Internal			
dlmb	SYS_Rst	sys_rst_out	I	Internal			
dlmb	LMB_Clk	sys_clk_s	I	Internal		CLK	
debug_mo...	DBG_TDO_0	DBG_TDO_0	I	Internal			
debug_mo...	DBG_UPDATE_0	DBG_UPDATE_0	O	Internal			
debug_mo...	DBG_CAPTURE_0	DBG_CAPTURE_0	O	Internal			
debug_mo...	DBG_REG_EN_0	DBG_REG_EN_0	O	Internal			
debug_mo...	DBG_CLK_0	DBG_CLK_0	O	Internal			
debug_mo...	OPB_Clk	sys_clk_s	I	Internal		CLK	
debug_mo...	DBG_TDI_0	DBG_TDI_0	O	Internal			
dcm_modul...	CLKIN	sys_clk_s	I	Internal			
dcm_modul...	CLKFB	zbt_dcm_feedback	I	Internal			
dcm_modul...	RST	zbt_dcm_rst	I	Internal			
dcm_modul...	CLK0	zbt_dcm_clk	O	Internal			
dcm_modul...	DSSEN	net_gnd	I	Internal			
dcm_modul...	PSEN	net_gnd	I	Internal			
dcm_modul...	LOCKED	zbt_dcm_locked	O	Internal			
clk_align_0	dcm_reset	zbt_dcm_rst	O	Internal			
clk_align_0	dcm0_locked	zbt_dcm_locked	I	Internal			
clk_align_0	external_clk	sys_clk_s	I	Internal		CLK	
clk_align_0	dcm1_locked	net_vcc	I	Internal			
clk_align_0	extend_dcm_reset	sys_rst_s	I	Internal			
clk_align_0	fpga_reset	sys_rst_out	O	Internal			
ZBT_512Kx32	Mem_ADV_LDN	ZBT_512Kx32_Mem_ADV_LDN	O	External			
ZBT_512Kx32	Mem_WEN	ZBT_512Kx32_Mem_WEN	O	External			
ZBT_512Kx32	Mem_A	ZBT_512Kx32_Mem_A	O	External	[0:31]		
ZBT_512Kx32	Mem_BEN	ZBT_512Kx32_Mem_BEN	O	External	[0:3]		
ZBT_512Kx32	Mem_OEN	ZBT_512Kx32_Mem_OEN	O	External	[0:0]		
ZBT_512Kx32	Mem_CEN	ZBT_512Kx32_Mem_CEN	O	External	[0:0]		
ZBT_512Kx32	Mem_CKEN	ZBT_512Kx32_Mem_CKEN	O	External			
ZBT_512Kx32	Mem_DQ	ZBT_512Kx32_Mem_DQ	IO	External	[0:31]		
ZBT_512Kx32	OPB_Clk	sys_clk_s	I	Internal		CLK	
RS232_1	TX	RS232_1_TX	O	External			
RS232_1	OPB_Clk	sys_clk_s	I	Internal		CLK	
RS232_1	RX	RS232_1_RX	I	External			
RS232	OPB_Clk	sys_clk_s	I	Internal		CLK	
RS232	TX	RS232_TX	O	External			
RS232	RX	RS232_RX	I	External			

External Port Connections: .UCF File

Net sys_clk PERIOD = 37037 ps;
Net sys_clk LOC=AH15;
Net sys_rst LOC=D10;

```
Net RS232_RX LOC=C8;
Net RS232_TX LOC=C9;
Net RS232_req_to_send LOC=B8;

Net opb_gpio_0_GPIO_IO<1> LOC=B27;
Net opb_gpio_0_GPIO_IO<0> LOC=F14;

Net PHY_slew1 LOC=G16;
Net PHY_slew2 LOC=C16;

Net opb_ethernet_0_PHY_crs LOC=F20;
Net opb_ethernet_0_PHY_col LOC=C23;
Net opb_ethernet_0_PHY_tx_data<3> LOC=C22;
Net opb_ethernet_0_PHY_tx_data<2> LOC=B20;
Net opb_ethernet_0_PHY_tx_data<1> LOC=B21;
Net opb_ethernet_0_PHY_tx_data<0> LOC=G20;
Net opb_ethernet_0_PHY_tx_en LOC=G19;
Net opb_ethernet_0_PHY_tx_clk LOC=H16;
Net opb_ethernet_0_PHY_tx_er LOC=D21;
Net opb_ethernet_0_PHY_rx_er LOC=D22;
Net opb_ethernet_0_PHY_rx_clk LOC=C17;
Net opb_ethernet_0_PHY_dv LOC=B17;
Net opb_ethernet_0_PHY_rx_data<0> LOC=B16;
Net opb_ethernet_0_PHY_rx_data<1> LOC=F17;
Net opb_ethernet_0_PHY_rx_data<2> LOC=F16;
Net opb_ethernet_0_PHY_rx_data<3> LOC=D16;
Net opb_ethernet_0_PHY_Mii_clk LOC=D17;
Net opb_ethernet_0_PHY_Mii_data LOC=A17;

Net ZBT_512Kx32_Mem_A<29> LOC=T23;
Net ZBT_512Kx32_Mem_A<29> FAST;
Net ZBT_512Kx32_Mem_A<28> LOC=U23;
Net ZBT_512Kx32_Mem_A<28> FAST;
Net ZBT_512Kx32_Mem_A<27> LOC=AB29;
Net ZBT_512Kx32_Mem_A<27> FAST;
Net ZBT_512Kx32_Mem_A<26> LOC=AA29;
Net ZBT_512Kx32_Mem_A<26> FAST;
Net ZBT_512Kx32_Mem_A<25> LOC=AA27;
Net ZBT_512Kx32_Mem_A<25> FAST;
Net ZBT_512Kx32_Mem_A<24> LOC=AB27;
Net ZBT_512Kx32_Mem_A<24> FAST;
Net ZBT_512Kx32_Mem_A<23> LOC=H25;
Net ZBT_512Kx32_Mem_A<23> FAST;
Net ZBT_512Kx32_Mem_A<22> LOC=G25;
Net ZBT_512Kx32_Mem_A<22> FAST;
Net ZBT_512Kx32_Mem_A<21> LOC=G28;
Net ZBT_512Kx32_Mem_A<21> FAST;
Net ZBT_512Kx32_Mem_A<20> LOC=H29;
Net ZBT_512Kx32_Mem_A<20> FAST;
Net ZBT_512Kx32_Mem_A<19> LOC=U27;
Net ZBT_512Kx32_Mem_A<19> FAST;
Net ZBT_512Kx32_Mem_A<18> LOC=T27;
Net ZBT_512Kx32_Mem_A<18> FAST;
```

```
Net ZBT_512Kx32_Mem_A<17> LOC=V29;
Net ZBT_512Kx32_Mem_A<17> FAST;
Net ZBT_512Kx32_Mem_A<16> LOC=U29;
Net ZBT_512Kx32_Mem_A<16> FAST;
Net ZBT_512Kx32_Mem_A<15> LOC=T24;
Net ZBT_512Kx32_Mem_A<15> FAST;
Net ZBT_512Kx32_Mem_A<14> LOC=T25;
Net ZBT_512Kx32_Mem_A<14> FAST;
Net ZBT_512Kx32_Mem_A<13> LOC=U28;
Net ZBT_512Kx32_Mem_A<13> FAST;
Net ZBT_512Kx32_Mem_A<12> LOC=F28;
Net ZBT_512Kx32_Mem_A<12> FAST;
Net ZBT_512Kx32_Mem_A<11> LOC=L23;
Net ZBT_512Kx32_Mem_A<11> FAST;
Net ZBT_512Kx32_Mem_DQ<31> LOC=T30;
Net ZBT_512Kx32_Mem_DQ<31> FAST;
Net ZBT_512Kx32_Mem_DQ<30> LOC=P28;
Net ZBT_512Kx32_Mem_DQ<30> FAST;
Net ZBT_512Kx32_Mem_DQ<29> LOC=R25;
Net ZBT_512Kx32_Mem_DQ<29> FAST;
Net ZBT_512Kx32_Mem_DQ<28> LOC=R29;
Net ZBT_512Kx32_Mem_DQ<28> FAST;
Net ZBT_512Kx32_Mem_DQ<27> LOC=R27;
Net ZBT_512Kx32_Mem_DQ<27> FAST;
Net ZBT_512Kx32_Mem_DQ<26> LOC=R23;
Net ZBT_512Kx32_Mem_DQ<26> FAST;
Net ZBT_512Kx32_Mem_DQ<25> LOC=N30;
Net ZBT_512Kx32_Mem_DQ<25> FAST;
Net ZBT_512Kx32_Mem_DQ<24> LOC=K26;
Net ZBT_512Kx32_Mem_DQ<24> FAST;
Net ZBT_512Kx32_Mem_DQ<23> LOC=M25;
Net ZBT_512Kx32_Mem_DQ<23> FAST;
Net ZBT_512Kx32_Mem_DQ<22> LOC=J29;
Net ZBT_512Kx32_Mem_DQ<22> FAST;
Net ZBT_512Kx32_Mem_DQ<21> LOC=K27;
Net ZBT_512Kx32_Mem_DQ<21> FAST;
Net ZBT_512Kx32_Mem_DQ<20> LOC=L24;
Net ZBT_512Kx32_Mem_DQ<20> FAST;
Net ZBT_512Kx32_Mem_DQ<19> LOC=H27;
Net ZBT_512Kx32_Mem_DQ<19> FAST;
Net ZBT_512Kx32_Mem_DQ<18> LOC=H26;
Net ZBT_512Kx32_Mem_DQ<18> FAST;
Net ZBT_512Kx32_Mem_DQ<17> LOC=K25;
Net ZBT_512Kx32_Mem_DQ<17> FAST;
Net ZBT_512Kx32_Mem_DQ<16> LOC=H28;
Net ZBT_512Kx32_Mem_DQ<16> FAST;
Net ZBT_512Kx32_Mem_DQ<15> LOC=J25;
Net ZBT_512Kx32_Mem_DQ<15> FAST;
Net ZBT_512Kx32_Mem_DQ<14> LOC=J26;
Net ZBT_512Kx32_Mem_DQ<14> FAST;
Net ZBT_512Kx32_Mem_DQ<13> LOC=J28;
Net ZBT_512Kx32_Mem_DQ<13> FAST;
Net ZBT_512Kx32_Mem_DQ<12> LOC=K24;
```

```
Net ZBT_512Kx32_Mem_DQ<12> FAST;
Net ZBT_512Kx32_Mem_DQ<11> LOC=J27;
Net ZBT_512Kx32_Mem_DQ<11> FAST;
Net ZBT_512Kx32_Mem_DQ<10> LOC=K29;
Net ZBT_512Kx32_Mem_DQ<10> FAST;
Net ZBT_512Kx32_Mem_DQ<9> LOC=L25;
Net ZBT_512Kx32_Mem_DQ<9> FAST;
Net ZBT_512Kx32_Mem_DQ<8> LOC=L26;
Net ZBT_512Kx32_Mem_DQ<8> FAST;
Net ZBT_512Kx32_Mem_DQ<7> LOC=P30;
Net ZBT_512Kx32_Mem_DQ<7> FAST;
Net ZBT_512Kx32_Mem_DQ<6> LOC=P23;
Net ZBT_512Kx32_Mem_DQ<6> FAST;
Net ZBT_512Kx32_Mem_DQ<5> LOC=P27;
Net ZBT_512Kx32_Mem_DQ<5> FAST;
Net ZBT_512Kx32_Mem_DQ<4> LOC=T29;
Net ZBT_512Kx32_Mem_DQ<4> FAST;
Net ZBT_512Kx32_Mem_DQ<3> LOC=R24;
Net ZBT_512Kx32_Mem_DQ<3> FAST;
Net ZBT_512Kx32_Mem_DQ<2> LOC=R28;
Net ZBT_512Kx32_Mem_DQ<2> FAST;
Net ZBT_512Kx32_Mem_DQ<1> LOC=U30;
Net ZBT_512Kx32_Mem_DQ<1> FAST;
Net ZBT_512Kx32_Mem_DQ<0> LOC=T28;
Net ZBT_512Kx32_Mem_DQ<0> FAST;
Net ZBT_512Kx32_Mem_BEN<0> LOC=G29;
Net ZBT_512Kx32_Mem_BEN<0> FAST;
Net ZBT_512Kx32_Mem_BEN<1> LOC=F29;
Net ZBT_512Kx32_Mem_BEN<1> FAST;
Net ZBT_512Kx32_Mem_BEN<2> LOC=H24;
Net ZBT_512Kx32_Mem_BEN<2> FAST;
Net ZBT_512Kx32_Mem_BEN<3> LOC=J24;
Net ZBT_512Kx32_Mem_BEN<3> FAST;
Net ZBT_512Kx32_Mem_WEN LOC=F26;
Net ZBT_512Kx32_Mem_WEN FAST;
Net ZBT_512Kx32_Mem_OEN<0> LOC=F30;
Net ZBT_512Kx32_Mem_OEN<0> FAST;
Net ZBT_512Kx32_Mem_CEN<0> LOC=G26;
Net ZBT_512Kx32_Mem_CEN<0> FAST;
Net ZBT_512Kx32_Mem_CKEN LOC=G30;
Net ZBT_512Kx32_Mem_CKEN FAST;
Net ZBT_512Kx32_Mem_ADV_LDN LOC=K23;
Net ZBT_512Kx32_Mem_ADV_LDN FAST;
Net ZBT_512Kx32_EMC_CLK_OUT LOC=G27;
Net ZBT_512Kx32_EMC_CLK_OUT FAST;


NET ZBT_512Kx32_EMC_CLK_FEEDBACK_IN LOC=AE15;
NET ZBT_512Kx32_EMC_CLK_FEEDBACK_IN FAST;
NET ZBT_512Kx32_EMC_CLK_FEEDBACK_OUT LOC=AH14;
NET ZBT_512Kx32_EMC_CLK_FEEDBACK_OUT FAST;
```

Simple FSL Communication Protocol

The Dual MicroBlaze system consists of two MicroBlaze systems as described in the hardware configuration section. From a software point of view, **microblaze_0** deals with all the communication protocol whereas **microblaze_1** deals with some fragmentation capability and is connected to **microblaze_0** with the FSLs. FSLs are unidirectional point to point communication buses. The role of the FSLs is to allow the two MicroBlazes exchange frame payloads that are sent or received by the node as well as some extra information needed for synchronization.

Due to the following characteristics of the FSLs, we had to design a special purpose communication mechanism to allow for proper transfer of data between the two MicroBlazes:

- *32 bit FSL transfers*: since the user will be always transferring payloads to transfer over Ethernet, or **microblaze_0** will be transferring payload of received frames as well as some demultiplexing data to **microblaze_1**, we had to find a way to tell the transfer's receiver how many bytes to wait for, so that it reads a finite number of 32 bit units, and disregards any bytes that were only introduced as a padding to the actual effective transfer content.

- *Blocking Operation*: There are two types of FSLs transfers, blocking transfers and non blocking transfers. The problem with blocking transfers is that they stop the whole pipeline from operation and the whole MicroBlaze system simply stops from doing anything until 4 bytes are read and the blocking operation ends. This creates many problems since interrupts are no longer operational.

In order to solve these problems, a simple special purpose FSL communication protocol was created and consists of the following ideas:

For **microblaze_0**, before starting any transfer, which occur after a valid frame is received by the Emac module, the following data is sent, followed by the frame's payload:

1. The Message Total Size: This information is taken from the layer 2 header.
2. The Source MAC Address: This is read from the frame header.
3. The Frame Type: This field is taken from the layer 2 header.

microblaze_1 keeps uses this information to update local tables and keep track of some parameters that allows it to know how many blocking reads to perform and how many bytes to read from these 32 bit units it is reading, as well as to provide the user with some variables used for demultiplexing of the frame payload, which directly follows this *control message*.

FSL Control Message: microblaze_0 → microblaze_1

Message Total Size	Source MAC Address	Frame Type	Payload
4 bytes	**6 bytes**	**2 bytes**	**Received Frame Payload**

For **microblaze_1**, before starting any transfer, which occurs when a user wants to send data to a second node on the network, the following *control message* is sent, followed by the payload of the message:

1. The Message Total Size: This information is taken from the layer 2 header.
2. The Destination MAC Address: the sender application will send this to notify microblaze_0 of the destination of the payload it is sending.

FSL Control Message: microblaze_1 → microblaze_0

Message Total Size	Destination MAC Address	Payload
4 bytes	**6 bytes**	**User Message to Send**

When the user sends a message that needs fragmentation, the FSLs will be reserved until the whole operation succeeds or is aborted due to a fragment which exceeded its allowable retransmission number. In this case the whole system is in *sending* mode and will simply reject all frames that arrive except for ACK frames. This was done to ensure that the FSLs always contain consistent data. Between sending two successive frames, microblaze_1 waits for an ACK transfer or a negative ACK transfer from microblaze_0 to know whether it transfers the payload of the next fragment to send or just aborts.

This provides a simple arbitration mechanism to ensure that the FSLs content is always read for the right reason and a function never reads what was intended to be read by another function.

Small Communication System with the Dual MicroBlaze Version and Network Monitoring Using the Sniffer

Introduction*:*

In this small experiment, you will be introduced on how to configure two boards for communication.

- Selection of MAC Addresses for each board.
- At the Sender: Using the Send_EMAC function and verifying the success or failure of the operation.
- At the Receiver: Using the receive handler function and some global variables to demultiplex the incoming frame payload.
- Using the Sniffer: A third board will be used as a sniffer to monitor network transactions.

Preparation*:*

Make sure you read the documentation on the dual MicroBlaze System as well as that on the protocol description before you start with the setup.

Equipment Needed*:*

- Two Multimedia Boards, or three in case you want to use a sniffer node.
- A Hub or a Switch and appropriate Ethernet RJ Cables. Note that you should pay attention of *not* using crossed cables! In case you are using a sniffer node, you should use a hub.

Setup*:*

12. In case you want to use a sniffer (Recommended for this experiment), read the document "Setting up the Sniffer Node" and perform the steps listed under *setup.* After completing this step, the sniffer should be operational on the network.

13. Go to your **DUAL_MB_EMAC** directory. Then go to the **APP1** directory. In the **APP1** directory double click on the XPS Icon.

14. Under Application Tab, expand the *Sources* Submenu of microblaze_0 and open the source code. As a user, you are not concerned with microblaze_0.

15. Do the same with the source code of microblaze_1. When you open the source for **mb_1.c** look for the section in **main** that lies between: "this is where your code starts" and "This is where your code ends". This will be the place where you will place your processing code as well as your **Send_EMAC** calls to send messages of any size reliably. A small test application has been written, take a look at it.

16. Without closing the XPS window, go to your **DUAL_MB_EMAC** Directory. Then go to the **APP2** directory. In the **APP2** directory double click on the XPS Icon.

17. Under the Application Tab, open the source for microblaze_0 and microblaze_1. Take a look at the section in **main** of **mb1.c** reserved for user code. It is empty because this board would act as a receiver and reception happens through a handler and not in the main function.

18. Connect the two boards to the Hub where the sniffer should already be connected and working.

MAC Addresses: Understanding how to assign MAC Addresses and what modifications to make in the code

1. Go to the following directory: **DUAL_MB_EMAC/APP1/code**. There open the **MACAddresses.h** File. This file will be read by both **mb0.c** and **mb1.c**.

2. The *NUMBER_OF_BOARDS* variable contains the number of definitions of Addresses just below it. Next you have Global Definitions of Addresses. The Local Address Pointer points to the local address of the board where the code will be downloaded. Note that Addresses should be unique, meaning that no two boards should have the same Address.

3. In this configuration, we can have a maximum of 5 boards on the network, sniffer not included. In case you need more boards, simply modify the *NUMBER_OF_BOARDS* variable and define some new addresses as the first 5 were defined. Again, the *LocalAddress* variable should be set appropriately.

4. For this test we only need two boards so keep the current *LocalAddress* as it originally was.

5. Now check the **`MACAddresses.h`** file in the **`DUAL_MB_EMAC/APP2/code`** just to know the local address of the receiver of the messages in this tutorial.

6. Take a look at the **`MACAddresses.c`** file. The first function just makes a comparison between two addresses and returns XST_SUCCESS or XST_FAILURE. The second function takes a MAC Address and returns an index that will be used as an index in an array of structures. You will need to modify this function in case you defined new addresses.

Sending Frames

In the **`APP1`** XPS Window, look at the source code for microblaze_1, in the user code section, a small test program has been created to show you how to send a frame, read it and make sure you understand it since it will help you understand the remaining part of the experiment.

Reception of Frames: Looking at the Handler Function

1. Open the **`mb1_HandlerFunctions.c`** file in the **`DUAL_MB_EMAC/APP2/code`**.

2. Search for the *void Handler(void * CallBackRef)* function in the code. In this function you should see a section between "this is where your code starts" and "This is where your code ends". In this section of code, you will read some variables to demultiplex the frame payload that will be contained in the *Receive_from_EMAC_Buffer*. Read well the comments to understand how to use the variables there to demultiplex the frame payload and copy it into your user's defined buffer.

Downloading and Testing the Communication

Remember that we have the Sniffer Node already monitoring the network. For the moment keep the serial port connected to it.

1. We will obviously start by downloading the receiver.

2. In the **`APP2`** XPS window, build the user's application, update the bitstream and then download it on the board. Since the microblaze_0's code is running on external memory (ZBTs) due to the size of the code, it has to be downloaded manually. Open an XMD Window, connect to the stub and download the executable of microblaze_0 to the board. (Instead of doing all that you can just type *source pat*, a user defined batch file), then type *run* to start the program. In case you get an error in this process, simply download the bitstream again, and then reopen an XMD window and try again.

3. Just after the previous step succeeds, the receiver is ready and the user led on the board should be turned on.

4. Now we need to download the Sender's bitstream on the board. Open the **`APP1`** XPS Window and do the same procedure as for **`APP2`**. As soon as the program starts to run look at the Terminal Window connected to the Sniffer, you will see some information about the frames traveling through the Hub, these are the Messages frames from app1 to app2 and the ack frames from app2 to app1.

5. Wait until the two messages appear on your screen (remember that there are several seconds of delay between them as we programmed in **`mb1.c`** of app1)

6. Redo the same test three other times: during the first test, keep the serial port on the Sniffer Board but toggle the user switch to change what to display. On the Second Test, place the serial port on the sender's board to check the Success Messages, and on the third test place the serial port on the receiver's side to check the variables and the data consistency test.

7. Try to modify the code to get more familiar with the system, for instance try to send a message to a board not present (like address1), try to modify the MAC Addresses, defining new MAC Addresses…
   NOTE: Read the "Some Important Notes" document before doing so.

Software Configuration and Source Code

MACAddresses.h

```
/*****************************************************************************
*
*       Simple Reliable Communication System over Ethernet _ Microblaze 0
*
*       MAC Addresses definitions, as well as local address selection.
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher-John Comis (University of Toronto)
*       at the following email address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/



#ifndef MACADDRESSES_H
#define MACADDRESSES_H
```

```
/************************** Number of Boards specification ***************************/

#define NUMBER_OF_BOARDS                5


/************************** MAC Addresses Definitions ********************************/


static Xuint8 Address0[6] =             //  48 bits  address
{
  0x06, 0x05, 0x04, 0x03, 0x02, 0x00
};

static Xuint8 Address1[6] =             //  48 bits  address
{
  0x06, 0x05, 0x04, 0x03, 0x02, 0x01
};

static Xuint8 Address2[6] =   //  48 bits  address
{
  0x06, 0x05, 0x04, 0x03, 0x02, 0x02
};

static Xuint8 Address3[6] =   //  48 bits  address
{
  0x06, 0x05, 0x04, 0x03, 0x02, 0x03
};

static Xuint8 Address4[6] =   //  48 bits  address
```

```
{
   0x06, 0x05, 0x04, 0x03, 0x02, 0x04
};

/****************************** MAC ADDRESS SELECTION ***********************************/

Xuint8 * LocalAddress = Address2;              // Choose our local Address Here, the rest of the setup is automatic

/****************************** MAC Address Functions definitions ***********************/

/*
* This function takes a pointer to two MAC Addresses and
* compares them
* returns XST_SUCCESS if they match
*/

XStatus CompareMACAddresses(Xuint8 *MAC1, Xuint8 *MAC2);



/*
* This function returns the appropriate index for the CommVars Array
* according to the MACAddress Field passed to it
* returns -1 for FAILURE
*/
Xuint32 GetIndex(Xuint8 *MACAddress);
#endif
```

MACAddresses.c

```
/*****************************************************************************
*
*       Simple Reliable Communication System over Ethernet _ Microblaze 0
*
*       This code contains the source of functions to perform comparison on MAC Addresses
*       as well as to return the index corresponding to a MAC Address in the structure used
*       to keep communications variables (Take a look at mb0_GlobalVariables.h and
*       mb1_GlobalVariables.h)
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher-John Comis (University of Toronto)
*       at the following email address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/

#include "MACAddresses.h"

/*
* This function takes a pointer to two MAC Addresses and
* compares them
```

```
* returns XST_SUCCESS if they match
*/

XStatus CompareMACAddresses(Xuint8 *MAC1, Xuint8 *MAC2)
{
        Xuint8 cnt;
        Xuint8 *ptr1 = MAC1;
        Xuint8 *ptr2 = MAC2;

        for(cnt = 0 ; cnt < 6 ; cnt++)
        {
                if( *(Xuint8 *)ptr1  !=   *(Xuint8 *)ptr2 )
                        return XST_FAILURE;

                ptr1 ++;
                ptr2 ++;

        }
        return XST_SUCCESS;
}



/*
* This function returns the appropriate index for the CommVars Array
* according to the MACAddress Field passed to it
* returns -1 for FAILURE
*/
Xuint32 GetIndex(Xuint8 *MACAddress)
{
```

```
	if(CompareMACAddresses((Xuint8*)Address0, (Xuint8 *)MACAddress) == XST_SUCCESS )
	{
		return 0 ;
	}
	else if(CompareMACAddresses((Xuint8*)Address1, (Xuint8 *)MACAddress) == XST_SUCCESS )
	{
		return 1 ;
	}
	else if(CompareMACAddresses((Xuint8*)Address2, (Xuint8 *)MACAddress) == XST_SUCCESS )
	{
		return 2 ;
	}
	else if(CompareMACAddresses((Xuint8*)Address3, (Xuint8 *)MACAddress) == XST_SUCCESS )
	{
		return 3 ;
	}
	else if(CompareMACAddresses((Xuint8*)Address4, (Xuint8 *)MACAddress) == XST_SUCCESS )
	{
		return 4 ;
	}
	else
		return -1;   // for errors
}
```

mb0.c

```
/*****************************************************************************
*
*       Simple Reliable Communication System over Ethernet _ Microblaze 0
*
*       This code setups the microblaze_0 system to deal with reliability
*       of communication with other boards
*
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher-John Comis (University of Toronto)
*       at the following email address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/

/*****************************  Include Files *********************************/

#include "xemac.h"
#include "xparameters.h"
#include "xstatus.h"
```

```
#include "mb_interface.h"       // MicroBlaze and FSLs
#include "xintc.h"                      // interrupt control
#include "xgpio.h"                      // DIP switch and output Led
#include "xtmrctr.h"                    // delayed frame retransmissions

#include "MACAddresses.h"
#include "mb0_GlobalVariables.h"   // contains the global variables shared among the 5 files
#include "mb0_HandlerFunctions.h"                       // contains the handlers functions
#include "mb0_DebugFunctions.h"             // contains functions for debugging using UArt
#include "mb0_InitializationFunctions.h"                                        // contains the functions that initialize our instances
#include "mb0_SendFunctions.h"              // contains the send functions form low level until reliable send
#include "mb0_FSLTransferFunctions.h"         // contains the functions to transfer data via the FSL link

#include "MACAddresses.c"
#include "mb0_HandlerFunctions.c"                       // contains the handlers functions
#include "mb0_DebugFunctions.c"             // contains functions for debugging using UArt
#include "mb0_InitializationFunctions.c"                                        // contains the functions that initialize our instances
#include "mb0_SendFunctions.c"              // contains the send functions form low level until reliable send
#include "mb0_FSLTransferFunctions.c"         // contains the functions to transfer data via the FSL link


/*********************************    MAIN FUNCTION      *******************************/

int main()
{
        /* Definitions */
        Xuint32 read_value;
        Xuint32 read_val;                       // to read from the fsl when we need to
        Xuint32 in_flag;                // to enter the function the first time
```

```
/* Initializations */
Timer_Expired = 1;
in_flag = 1;
BusArbitration = 0;                    // initialized to read and send a frame

/* MicroBlaze Parametrization */
Initialize_Parameters();               // initializes the whole system for this microblaze

/* Synchronization with the second microblaze
  So that they enter the "main" together                */

microblaze_bwrite_datafsl(10,0);
microblaze_bread_datafsl(read_value, 1);

/* Now after synchronization
  this microblaze is ready to operate and transfer data        */

/* Loop for reading data from the second microblaze
  and transfer it to EMAC controller                                    */
while(1)
{
        /* Waiting for the Wake-Up byte sent by the second microblaze
          We use a looping non blocking operation to be able to be interrupted by EMAC */

        while(in_flag == 1)
        {
                microblaze_nbread_datafsl(read_val, 1);
                asm("addc %0, r0, r0": "=d"(in_flag));

                if(in_flag != 1)
```

```
                {
                        BusArbitration = 1;
                        /* We received a Wake-Up Byte from Application MicroBlaze               */
                }
        }
        /* Now get the control message to know what to expect
          Then get the message to send and send it reliably over EMAC      */

    Get_App_Send_EMAC_Message();

        /*      Now Microblaze_0 is allowed to interrupt the second microblaze to
                check if it can send to it data */
        BusArbitration = 0;

        /* Now we need to wait again for a Wake-Up Byte */
        in_flag = 1;
    }
    return 0;
}
```

mb0_DebugFunctions.h

```
/*****************************************************************************
*
*       Simple Reliable Communication System over Ethernet _ Microblaze 0
*
*       This code contains the definitions of various functions used for debugging
*       as well as for returning frame and message fields and perform MAC Address operations
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher-John Comis (University of Toronto)
*       at the following email address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/

#ifndef MB0_DEBUGFUNCTIONS_H
#define MB0_DEBUGFUNCTIONS_H

#include "mb0_GlobalVariables.h"

/*
*   returns the character equivalent of an integer
```

```
*/
char map(int x);

/*
* Pass it a Pointer to the start of the Ethernet Header
* Returns the Layer 2 Type Field
*/
Xuint16 GetType(Xuint8 *MsgBuffer);

/*
* Pass it a Pointer to the start of the Ethernet Header
* Returns the Layer 2 Sequence Number Field
*/
Xuint32 GetSeqNum(Xuint8 *MsgBuffer);

/*
* Pass it a Pointer to the start of the Ethernet Header
* Returns the Layer 2 Message Size Field
*/
Xuint32 GetSize(Xuint8 *MsgBuffer);


/*
* Pass it a pointer to the EMAC Address
* Fills the global variable 'array' with
* the ETHERNET Address in standard format
*
* array is printed using xil_printf("%s", array);
*/
void fill_array(Xuint8 *Ptr);
```

```
/*
* Prints the EMAC Statistics
*
* NOTE: RS232 has to be set on microblaze_0 to view what
*                 these functions are displaying
*/
void PrintEmacStats(XEmac_Stats Stats);

/*
* Pass it a pointer to the Ethernet Frame
*
* Prints the Layer 2 Variables
*
* Message Header Format:    Type                    _ 2 bytes
*                                                   Sequence Number      _ 4 bytes
*                   Message Length     _ 4 bytes
*
* NOTE: RS232 has to be set on microblaze_0 to view what
*                 these functions are displaying
*/
void Print_Message_Fields(Xuint8 *MsgBuffer, int MsgLen);

/*
* Pass it a pointer to the Ethernet frame
*
* and the MsgLen as returned by the ReceiveFifo function
*
* Ethernet Header Format:    Destination Address _ 6 bytes
*                                                   Source Address     _ 6 bytes
*                   Len / Type Field    _ 2 bytes
```

```
*
* NOTE: RS232 has to be set on microblaze_0 to view what
*                these functions are displaying
*/
void Print_Frame_Fields(Xuint8 *MsgBuffer, int MsgLen);

#endif
```

mb0_DebugFunctions.c

```
/*****************************************************************************
*
*       Simple Reliable Communication System over Ethernet _ Microblaze 0
*
*       This code contains definitions of various functions that deal with
*       communications via the fsl
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher-John Comis (University of Toronto)
*       at the following email address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/
```

```
#include "mb0_GlobalVariables.h"
#include "mb0_DebugFunctions.h"

/*
*   returns the character equivalent of an integer
*/
char map(int x)
{
        switch(x)
        {
        case 0:
                return '0';
        case 1:
                return '1';
        case 2:
                return '2';
        case 3:
                return '3';
        case 4:
                return '4';
        case 5:
                return '5';
        case 6:
                return '6';
        case 7:
                return '7';
        case 8:
                return '8';
        case 9:
                return '9';
```

```
		case 10:
			return 'A';
		case 11:
			return 'B';
		case 12:
			return 'C';
		case 13:
			return 'D';
		case 14:
			return 'E';
		case 15:
			return 'F';
		}
}


/*
* Pass it a Pointer to the start of the Ethernet Header
* Returns the Layer 2 Type Field
*/
Xuint16 GetType(Xuint8 *MsgBuffer)
{
		Xuint8 * Ptr = MsgBuffer;
		Ptr+=14;
		return *(Xuint16 *)Ptr;
}

/*
* Pass it a Pointer to the start of the Ethernet Header
* Returns the Layer 2 Sequence Number Field
```

```
*/
Xuint32 GetSeqNum(Xuint8 *MsgBuffer)
{
        Xuint8 * Ptr = MsgBuffer;
        Ptr+=16;
        return *(Xuint32 *)Ptr;
}

/*
* Pass it a Pointer to the start of the Ethernet Header
* Returns the Layer 2 Message Size Field
*/
Xuint32 GetSize(Xuint8 *MsgBuffer)
{
        Xuint8 * Ptr = MsgBuffer;
        Ptr+=20;
        return *(Xuint32 *)Ptr;
}


/*
* Pass it a pointer to the EMAC Address
* Fills the global variable 'array' with
* the ETHERNET Address in standard format
*/
void fill_array(Xuint8 *Ptr)
{
        Xuint8 mycounter;
        for(mycounter = 0 ; mycounter < PRINT_MAC_ADDRESS_SIZE ; mycounter += 3)
        {
```

```
                array[mycounter+1] = map(*(Ptr) & 0xF);
                array[mycounter] = map( ( *(Ptr) & 0xF0 ) >> 4);
                Ptr++;
                if(mycounter<15) array[mycounter+2] = '-';
        }
}

/*
* Prints the EMAC Statistics
*
* NOTE: RS232 has to be set on microblaze_0 to view what
*               these functions are displaying
*/
void PrintEmacStats(XEmac_Stats Stats)                                          // for debugging
{
   xil_printf("Number of frames transmitted                  %d\n\r", Stats.XmitFrames);
//    xil_printf("Number of bytes transmitted                  %d\n\r", Stats.XmitBytes);
   xil_printf("Number of transmission failures due to late collisions   %d\n\r", Stats.XmitLateCollisionErrors);
   xil_printf("Number of transmission failures due to excess collision deferrals   %d\n\r", Stats.XmitExcessDeferral);
//    xil_printf("Number of transmit overrun errors   %d\n\r", Stats.XmitOverrunErrors);
//    xil_printf("Number of transmit underrun errors  %d\n\r", Stats.XmitUnderrunErrors);
   xil_printf("Number of frames received                %d\n\r", Stats.RecvFrames);
//    xil_printf("Number of bytes received                  %d\n\r", Stats.RecvBytes);
   xil_printf("Number of frames discarded due to FCS errors                  %d\n\r", Stats.RecvFcsErrors);
   xil_printf("Number of frames received with alignment errors      %d\n\r", Stats.RecvAlignmentErrors);
//    xil_printf("Number of frames discarded due to overrun errors   %d\n\r", Stats.RecvOverrunErrors);
//    xil_printf("Number of recv underrun errors        %d\n\r", Stats.RecvUnderrunErrors);
//    xil_printf("Number of frames missed by MAC   %d\n\r", Stats.RecvMissedFrameErrors);
   xil_printf("Number of frames discarded due to collisions           %d\n\r", Stats.RecvCollisionErrors);
   xil_printf("Number of frames discarded with invalid length field   %d\n\r", Stats.RecvLengthFieldErrors);
```

```
//    xil_printf("Number of short frames discarded    %d\n\r", Stats.RecvShortErrors);
//    xil_printf("Number of long frames discarded     %d\n\r", Stats.RecvLongErrors);
//    xil_printf("Number of DMA errors since init     %d\n\r", Stats.DmaErrors);
//    xil_printf("Number of FIFO errors since init    %d\n\r", Stats.FifoErrors);
//    xil_printf("Number of receive interrupts                %d\n\r", Stats.RecvInterrupts);
//    xil_printf("Number of transmit interrupts       %d\n\r", Stats.XmitInterrupts);
//    xil_printf("Number of MAC (device) interrupts   %d\n\r", Stats.EmacInterrupts);
//    xil_printf("Total interrupts                                %d\n\r", Stats.TotalIntrs);
}

/*
* Pass it a pointer to the Ethernet Frame
*
* Prints the Layer 2 Variables
*
* Message Header Format:    Type                    _ 2 bytes
*                                                   Sequence Number     _ 4 bytes
*                   Message Length     _ 4 bytes
*
* NOTE: RS232 has to be set on microblaze_0 to view what
*                these functions are displaying
*/
void Print_Message_Fields(Xuint8 *MsgBuffer, int MsgLen)
{
        Xuint32 mycounter;
        Xuint32 LenField;
        Xuint16 TypeofMess;
        Xuint32 FragmentPayloadLength; // since FragmentLength can be different
                                                  // than Message PayloadLength
        Xuint8 *MesgPtr = MsgBuffer;
```

```
Xuint8 *TemporaryPtr = (Xuint8 *)TemporaryBuffer;

xil_printf("Message Parameters\r\n");
xil_printf("_____\r\n");

MesgPtr+=14;
TypeofMess = *(Xuint16 *)MesgPtr;
xil_printf("Message Type Field                : %d\n\r", TypeofMess);

MesgPtr+=2;

xil_printf("Message Sequence Nb Field       : %d\n\r", *(Xuint32 *)MesgPtr);

MesgPtr+=4;
LenField = *(Xuint32 *)MesgPtr;
xil_printf("Message Length                   : %d\n\r", LenField);

MesgPtr+=4;

if(LenField <= 1490)
        FragmentPayloadLength = LenField;   // The Frame contains the Message header (10 bytes)
                                                                                // and the whole message itself as its payload
  // case of fragmented message
else
{
        if(TypeofMess == FIRST_FRAGMENT_TYPE || TypeofMess == INTERNAL_FRAGMENT_TYPE) // frame is full
                FragmentPayloadLength = MAX_L2_PAYLOAD;
        else if (TypeofMess == LAST_FRAGMENT_TYPE) // last fragment of a message
        {
                if(LenField % MAX_L2_PAYLOAD == 0) // 1490 payload size of last fragment
```

```
                        FragmentPayloadLength = MAX_L2_PAYLOAD;
                else
                        FragmentPayloadLength = (LenField % MAX_L2_PAYLOAD);
        }
    }

// Printing the first byte of the fragment payload
    xil_printf("Fragment Payload 1st byte: %x\n\r", *(Xuint8 *)MesgPtr);

// Getting to the last byte of the payload
    MesgPtr += (FragmentPayloadLength - 1);

// Printing the last byte of the fragment payload
    xil_printf("Fragment Payload last byte: %x\n\r", *(Xuint8 *)MesgPtr);

/*
    xil_printf("Frame Payload                    :\r\n");
    for(mycounter=0 ; mycounter < FragmentPayloadLength ; mycounter++)
    {
        xil_printf("%x", *(Xuint8 *)MesgPtr);
        MesgPtr++;
    }
*/
    xil_printf("\r\n_____________________________________________________\r\n");
}

/*
* Pass it a pointer to the Ethernet frame
*
* and the MsgLen as returned by the Receive function
```

```
*
* Ethernet Header Format:     Destination Address _ 6 bytes
*                                                   Source Address      _ 6 bytes
*                         Len / Type Field    _ 2 bytes
*
* NOTE: RS232 has to be set on microblaze_0 to view what
*                    these functions are displaying
*/
void Print_Frame_Fields(Xuint8 *MsgBuffer, int MsgLen)
{
        Xuint8 *MesgPtr = MsgBuffer;

        xil_printf("Frame Parameters\r\n");
        xil_printf("_____________________\r\n");

        xil_printf("Frame Length                    : %d\r\n", MsgLen);

        fill_array(MesgPtr);
        xil_printf("Destination MAC                 : %s\r\n", array);

        MesgPtr+=6;

        fill_array(MesgPtr);
        xil_printf("Source      MAC                 : %s\r\n", array);

        MesgPtr+=6;

        xil_printf("Payload Length                  : %d\r\n", *(Xuint16 *)MesgPtr);
        xil_printf("\r\n_____________________________________________________\r\n");
}
```

mb0_FSLTransferFunction.h

```
/*****************************************************************************
*
*       Simple Reliable Communication System over Ethernet _ Microblaze 0
*
*       This code contains definitions of various functions that deal with
*       communications via the fsl
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher-John Comis (University of Toronto)
*       at the following email address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/

#ifndef MB0_FSLTRANSFERFUNCTIONS_H
#define MB0_FSLTRANSFERFUNCTIONS_H

/*
*       Empty Handler
*/
void TimerCounterHandler2(void * CallBackRef);
```

```
/*
*       Turns on the user led, can be used for debugging
*       Note: in the Initialize_Parameters, the led will be turned on so this function
*       is only usefull if you disable the turning on of the led in the Initialize_GPio function
*/
void Turn_On_Led();


/*
*       Function that starts the timer counter 1 that will
*       trigger an interrupt that will be the input of the
*       interrupt controller of microblaze_1
*
*       Some synchronization is done so that we are sure we interrupted
*       microblaze_1
*
*       FUTURE WORK:    Try not to use the timer to do this, instead
*                               use a dedicated signal
*/
void Interrupt_Second_MicroBlaze();

/*
*       Takes as input the Total Message Size
*       returns the number of fragments needed
*       to transfer the message
*/
Xuint32 Get_Number_of_Fragments(Xuint32 TotalSize);

/*
```

```
* Takes the total message payload size
* returns the size of the last fragment payload
*/
Xuint32 Get_Last_Fragment_Size(Xuint32 TotalSz);

/*
*       Takes a pointer to the buffer where to write and the number of bytes to
*       read from the fsls
*       It will perform the appropriate number of FSL reads to get the number of bytes required
*/
void Get_from_App(Xuint8 *BuffPtr, Xuint32 Length_Bytes);


/*
*       Takes a pointer to the buffer where to read, the total length of the message,
*       the length of the frame payload we want to send to microblaze_1 and a pointer
*       to the MAC Address of the sender
*
*       It first sends a control message (Total Message Size and Source MAC Address)
*       Then It sends the payload of the frame
*/
void Send_Frame_to_App(Xuint8 * Receive_Buffer_Ptr, Xuint32 TotalMessageLength, Xuint32 FrameLength, Xuint8 * Addr, Xuint16 FrameType);


/*
*       Takes the Message Total Size, as well as a pointer to the MAC Address
*       It will send to microblaze_1 the following data:
*
*       Message Total Size (4 bytes) __ Source MAC Address (6 bytes) __ Fragment Type (2 bytes)
*
```

```
*/
void Send_to_App_Control_Message(Xuint32 MessageTotalSize, Xuint8 * AddressPointer, Xuint16 FragType);

/*
*       Takes a pointer to the buffer where to read as well as the number of bytes to
*       send to microblaze_1 via fsl transfers
*/
void Send_to_App(Xuint8 *BuffPtr, Xuint32 Length_Bytes);


/*
* Takes a pointer to the control message
* will set the global variables TotalMessageSize
* and DestinationAddress array
*
* by calling fill_array(pointer to the MAC Address location)
* we will fill 'array' with the EMAC printable address in standard format
* Use xil_printf("%s", array)
*/
void Parse_Control_Message(Xuint8 * BufferPointer);


/*
*       It first reads 10 bytes (the control message) to determine the Destination MAC Address
*       then it reads the message in chunks (fragments of maximum size 1490 bytes) and sends them
*       reliably to the Destination Address.
*
*       In case of failure, SendReliably returns XST_FAILURE and we exit the function
*
*       Note that we are notifying microblaze_1 of successes and failures after sending each fragment
```

```
*/
static Xuint8 Get_App_Send_EMAC_Message();

#endif
```

mb0_FSLTransferFunction.c

```
/*****************************************************************************
*
*       Simple Reliable Communication System over Ethernet _ Microblaze 0
*
*       This code contains the source code for inter microblaze communication functions
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher-John Comis (University of Toronto)
*       at the following email address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/

#include "mb0_FSLTransferFunctions.h"
```

```
void TimerCounterHandler2(void * CallBackRef)
{
	// does nothing since this is only to interrupt the second microblaze
}

/*
*	Turns on the user led, can be used for debugging
*	Note: in the Initialize_Parameters, the led will be turned on so this function
*	is only usefull if you disable the turning on of the led in the Initialize_GPio function
*/
void Turn_On_Led()
{
   XGpio_Initialize(&Gpio, XPAR_OPB_GPIO_0_DEVICE_ID);
	XGpio_SetDataDirection(&Gpio, 1,~LED);
}


/*
*	Function that starts the timer counter 1 that will
*	trigger an interrupt that will be the input of the
*	interrupt controller of microblaze_1
*
*	Some synchronization is done so that we are sure we interrupted
*	microblaze_1
*/
void Interrupt_Second_MicroBlaze()
{
	Xuint32 val;
	XTmrCtr_Start(&TimerCounter2, 0);		// send an interrupt signal
	microblaze_bread_datafsl(val, 1);			// get an ack message on the interrupt
```

```
	if(val == 1)		// meaning the second microblaze is in write mode
	{
		BusArbitration = 1;
	}
	else if (val == 0)	// meaning the second microblaze is in read mode
	{
		BusArbitration = 0;
	}

	XTmrCtr_Stop(&TimerCounter2, 0);			// we stop the counter
	microblaze_bwrite_datafsl(10,0);		// let it exit the handler after ack the INTC
}


/*****************************  Function Definitions  ***********************************/

/*
*	Takes as input the Total Message Size
*	returns the number of fragments needed
*	to transfer the message
*/
Xuint32 Get_Number_of_Fragments(Xuint32 TotalSize)
{
	if(TotalSize % MAX_L2_PAYLOAD == 0)
		return TotalSize / MAX_L2_PAYLOAD;
	else
		return 1 + TotalSize / MAX_L2_PAYLOAD;
}
```

```
/*
*	Takes a pointer to the buffer where to write and the number of bytes to
*	read from the fsls
*	It will perform the appropriate number of FSL reads to get the number of bytes required
*/
void Get_from_App(Xuint8 *BuffPtr, Xuint32 Length_Bytes)
{
	Xuint32 TempSize = Length_Bytes;
	Xuint32 Temp;

	Xuint8 *Ptr = BuffPtr;
	Xuint8 *Pointer;

	while(TempSize >= 4)
	{
		microblaze_bread_datafsl(Temp, 1);

		Pointer = (Xuint8 *)&Temp;

		*Ptr++ = *Pointer++;
		*Ptr++ = *Pointer++;
		*Ptr++ = *Pointer++;
		*Ptr++ = *Pointer++;

		TempSize = TempSize - 4; // read 4 bytes
	}
	if(TempSize == 3)
	{
		microblaze_bread_datafsl(Temp, 1);
```

```
			Pointer = (Xuint8 *)&Temp;

			*Ptr++ = *Pointer++;
			*Ptr++ = *Pointer++;
			*Ptr++ = *Pointer++;
		}
		else if(TempSize == 2)
		{
			microblaze_bread_datafsl(Temp, 1);

			Pointer = (Xuint8 *)&Temp;

			*Ptr++ = *Pointer++;
			*Ptr++ = *Pointer++;
		}
		else if(TempSize == 1)
		{
			microblaze_bread_datafsl(Temp, 1);

			Pointer = (Xuint8 *)&Temp;

			*Ptr++ = *Pointer++;
		}
}



/*
*	Takes a pointer to the buffer where to read, the total length of the message,
*	the length of the frame payload we want to send to microblaze_1 and a pointer
```

```
*	to the MAC Address of the sender
*
*	It first sends a control message (Total Message Size and Source MAC Address)
*	Then It sends the payload of the frame
*/
void Send_Frame_to_App(Xuint8 * Receive_Buffer_Ptr, Xuint32 TotalMessageLength, Xuint32 FrameLength, Xuint8 * Addr, Xuint16 FrameType)
{
	Xuint8 * Address_Ptr = Addr;

	// Sending a Control Message for notification about source of frame
	Send_to_App_Control_Message(TotalMessageLength, Address_Ptr, FrameType);  // has to know total length and fragment length

	// Sending the Frame Payload itself
	Send_to_App(Receive_Buffer_Ptr, FrameLength);
}


/*
*	Takes the Message Total Size, as well as a pointer to the MAC Address
*	It will send to microblaze_1 the following data:
*
*	Message Total Size (4 bytes) __ Source MAC Address (6 bytes) __ Fragment Type (2 bytes)
*
*/
void Send_to_App_Control_Message(Xuint32 MessageTotalSize, Xuint8 * AddressPointer, Xuint16 FragType)
{
	Xuint8 * Ptr = Control_Buffer;
	Xuint8 * AddrPtr = AddressPointer;

	*(Xuint32 *)Ptr = MessageTotalSize;		// Writing Message Total Size
```

```
                                                        // In Control Buffer
        Ptr += 4;

   *Ptr++ = *AddrPtr++;                         // Writing destination address
   *Ptr++ = *AddrPtr++;                         // In Control Buffer
   *Ptr++ = *AddrPtr++;
   *Ptr++ = *AddrPtr++;
   *Ptr++ = *AddrPtr++;
   *Ptr++ = *AddrPtr++;

        *(Xuint16 *)Ptr = FragType;             // Writing Message Type

        Ptr = Control_Buffer;                   // Setting back the pointer at the start

        // Now Sending the Content of Control Buffer
        microblaze_bwrite_datafsl(*(Xuint32 *)Ptr,0);
        Ptr += 4;
        microblaze_bwrite_datafsl(*(Xuint32 *)Ptr,0);
        Ptr += 4;
        microblaze_bwrite_datafsl(*(Xuint32 *)Ptr,0);
}


/*
*       Takes a pointer to the buffer where to read as well as the number of bytes to
*       send to microblaze_1 via fsl transfers
*/
void Send_to_App(Xuint8 *BuffPtr, Xuint32 Length_Bytes)
{
        Xuint32 Length_Words;
```

```
	Xuint32 * Ptr = (Xuint32 *)BuffPtr;
	Xuint32 counter;

	if(Length_Bytes % 4 == 0)
		Length_Words = Length_Bytes / 4;
	else
		Length_Words = 1 + Length_Bytes / 4;

	for(counter = 0 ; counter < Length_Words ; counter ++)
	{
		microblaze_bwrite_datafsl(*Ptr++, 0);
	}

	//	In case we read and send something from outside the buffer
	//  after byte 1490 we get no problem since the receiver would disregard any
	//	extra bytes after byte 1490
}


/*
* Takes a pointer to the control message
* will set the global variables TotalMessageSize
* and DestinationAddress array
*
* by calling fill_array(pointer to the MAC Address location)
* we will fill 'array' with the EMAC printable address in standard format
* Use xil_printf("%s", array)
*/
void Parse_Control_Message(Xuint8 * BufferPointer)
{
```

```
	Xuint8 * Ptr = BufferPointer;

	Total_Message_Size = *(Xuint32 *)Ptr;

	Ptr += 4;

	fill_array( Ptr );

	Destination_Address[0] = *Ptr++;
	Destination_Address[1] = *Ptr++;
	Destination_Address[2] = *Ptr++;
	Destination_Address[3] = *Ptr++;
	Destination_Address[4] = *Ptr++;
	Destination_Address[5] = *Ptr++;
}


/*
*	It first reads 10 bytes (the control message) to determine the Destination MAC Address
*	then it reads the message in chunks (fragments of maximum size 1490 bytes) and sends them
*	reliably to the Destination Address.
*
*	In case of failure, SendReliably returns XST_FAILURE and we exit the function
*
*	Note that we are notifying microblaze_1 of successes and failures after sending each fragment
*/
static Xuint8 Get_App_Send_EMAC_Message()
{
	XStatus Res;
	Xuint32 NbFrags;
```

```
Xuint32 CurrentFragmentSize;
Xuint32 counter;
Xuint32 TempSize;
Xuint32 LastFragSize;

Get_from_App(Send_Buffer, 10); // Get the Control Message.

Parse_Control_Message(Send_Buffer);

TempSize = Total_Message_Size;

NbFrags = Get_Number_of_Fragments(Total_Message_Size);

// Now we are ready to receive data in chunks and to place them in a temporary buffer
// before sending them using the Reliable Function to the destination

if(Total_Message_Size <= MAX_L2_PAYLOAD)
{
        Get_from_App(Send_Buffer, Total_Message_Size);

        Res = SendReliably(Total_Message_Size, (Xuint8 *) Send_Buffer, Destination_Address, NON_FRAGMENTED_MESSAGE_TYPE);

                if(Res == XST_SUCCESS)
                {
                        microblaze_bwrite_datafsl(0,0); // send an ack to second microblaze
                }
                else
                {
                        microblaze_bwrite_datafsl(1,0); // send an failure ack to second microblaze
                        return 1; // for failure
```

```
		}
}

else
{
	while(TempSize > 0)
	{
		if(TempSize == Total_Message_Size) // the first message fragment
		{
			TempSize = TempSize - MAX_L2_PAYLOAD;
			Get_from_App(Send_Buffer, MAX_L2_PAYLOAD);

			// effectively gets 1492 with upper two discarded

			Res = SendReliably(Total_Message_Size, (Xuint8 *) Send_Buffer, Destination_Address, FIRST_FRAGMENT_TYPE);

				if(Res == XST_SUCCESS)
				{
					microblaze_bwrite_datafsl(0,0); // send an ack to second microblaze
				}
				else
				{
					microblaze_bwrite_datafsl(1,0); // send an failure ack to second microblaze
					return 1; // for failure
				}

		}
		else if(TempSize > MAX_L2_PAYLOAD)  // for all the Fragments except the last one
		{
			TempSize = TempSize - MAX_L2_PAYLOAD;
```

```
                Get_from_App(Send_Buffer, MAX_L2_PAYLOAD);

                Res = SendReliably(Total_Message_Size, (Xuint8 *) Send_Buffer, Destination_Address,
INTERNAL_FRAGMENT_TYPE);
                        if(Res == XST_SUCCESS)
                        {
                                microblaze_bwrite_datafsl(0,0); // send an ack to second microblaze
                        }
                        else
                        {
                                microblaze_bwrite_datafsl(1,0); // send an failure ack to second microblaze
                                return 1; // for failure
                        }
        }
        else if(TempSize <= MAX_L2_PAYLOAD)
        {
                Get_from_App(Send_Buffer, TempSize);

                Res = SendReliably(Total_Message_Size, (Xuint8 *) Send_Buffer, Destination_Address, LAST_FRAGMENT_TYPE);

                        if(Res == XST_SUCCESS)
                        {
                                microblaze_bwrite_datafsl(0,0); // send an ack to second microblaze
                        }
                        else
                        {
                                microblaze_bwrite_datafsl(1,0); // send an failure ack to second microblaze
                                return 1; // for failure
                        }
```

```
                        TempSize = 0;
                    }
                }               // end of while
        }                       // end of else

        //      xil_printf("Success in Sending Totality\n\r");
        //      Send a Success Message

        return 0; // success
}


/*
* Takes the total message payload size
* returns the size of the last fragment payload
*/
Xuint32 Get_Last_Fragment_Size(Xuint32 TotalSz)
{
        if(TotalSz % MAX_L2_PAYLOAD == 0)
                return MAX_L2_PAYLOAD;
        else
                return TotalSz % MAX_L2_PAYLOAD;
}
```

mb0_GlobalVariables.h

```
/*****************************************************************************
*
*       Simple Reliable Communication System over Ethernet _ Microblaze 0
*
*       This code contains Global Variables and structures
*       used by microblaze_0
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher-John Comis (University of Toronto)
*       at the following email address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/


#ifndef MB0_GLOBALVARIABLES_H
#define MB0_GLOBALVARIABLES_H

#include "MACAddresses.h"

/***************************** Constants Definitions ***********************************/
```

```
#define XEM_MAX_FRAME_SIZE_IN_WORDS ((XEM_MAX_FRAME_SIZE / sizeof(Xuint32)) + 1)

#define TEST_1_OPTIONS          (XEM_UNICAST_OPTION | XEM_INSERT_PAD_OPTION | \
                    XEM_INSERT_FCS_OPTION | XEM_INSERT_ADDR_OPTION | \
                    XEM_OVWRT_ADDR_OPTION)

#define LED    0x1                          // bit 0 of GPIO is connected to the LED

#define INTC_INPUT_EMAC     0               // EMAC has higher priority (check add/edit cores)
#define INTC_INPUT_TIMER    1               // TIMER has lower priority

#define TIMER_COUNTER_0     0       // Timer Counter 0 is used
#define INTC_DEVICE_ID      0


#define ACK_L2_PAYLOAD_SIZE         36
#define ACK_L1_PAYLOAD_SIZE         46
#define ACK_FRAME_SIZE              60
#define L2_HEADER_SIZE              10
#define MAX_RETRANSMIT_ATTEMPTS     10

#define ACK_MESSAGE_TYPE            1
#define NON_FRAGMENTED_MESSAGE_TYPE 0
#define FIRST_FRAGMENT_TYPE         2
#define INTERNAL_FRAGMENT_TYPE      3
#define LAST_FRAGMENT_TYPE          4
#define MAX_L2_PAYLOAD              1490
#define PRINT_MAC_ADDRESS_SIZE      17
```

```
/************************** Instance Definitions ***********************************/

static XGpio Gpio;                  // one will be set as output led, the other as an input DIP switch
static XEmac Emac;
static XIntc InterruptController;   // input 0 (high priority) is from EMAC, input 1 is from timer_intr
static XTmrCtr TimerCounter;            // for retransmission
static XTmrCtr TimerCounter2;           // to interrupt second microblaze

XEmac *EmacPtr;                                 // Initialized in Init.h



/******************************** Buffers ******************************/

static Xuint32 TemporaryBuffer;     // To hold temporarily 32 bit values for alignment problems

static Xuint32 TxMessage[XEM_MAX_FRAME_SIZE_IN_WORDS + 40]; // holds fragment to send
static Xuint32 TxFrame[XEM_MAX_FRAME_SIZE_IN_WORDS];            // holds frame to send
static Xuint32 RecvBuffer[XEM_MAX_FRAME_SIZE_IN_WORDS];         // holds incoming frame

static Xuint8 AckFrame[60];                                                 // holds the ACK to send
static Xuint32 BusArbitration;                  // 0 for receiving mode, 1 for send mode


static Xuint8 Send_Buffer[1490];            // holds the payload as we get it from microblaze_1
static Xuint8 Control_Buffer[12];           // holds the control message
static Xuint8 Destination_Address[6];       // holds the destination address we get from the
                                                        // control buffer

/************************** Global Variables ***************************/
```

```
char array[PRINT_MAC_ADDRESS_SIZE];                    // used to hold MAC addresses in standard format

static Xuint32 Total_Message_Size;

XEmac_Stats Stats;          // One Variable for the EMAC Module.

Xuint8 Timer_Expired;       // turns to one everytime the timer expires


// retransmission reset values
        // RESET VALUE --> DELAY IN SECONDS
        // 0xF0000000  --> 10 seconds approx.
        // 0xF7000000  --> 5.5 seconds approx.
        // 0xFE600000  --> 1 second approx.
        // 0xFF400000  --> 0.5 seconds approx.

Xuint32 RstVal[10] = {  0xFE600000,0xFF000000,0xFF000000,
                                    0xFE000000 ,0xFE000000 ,0xFE000000 , 0xFD500000,
                                    0xFD000000, 0xFC700000 , 0xFC000000};

/************************** Communication Variables ***********************/

/*
*
* This structure contains all the variables to be used between
* Two Specific Nodes.
*
* This type of implementation allows us to have seperate reliable links
*
```

```
* For a system of 'N' Communication Nodes, each single node will have 'n - 1' structures,
* each one for communicating with one neighbor. (one more is the local structure but
* it serves to nothing)
*
*/

typedef struct
{

Xuint32 SeqNum;                                 // Holds Wrapping Off Sequence Numbers of Messages

Xuint32 WaitingFrameNumber;             // Holds the Frame Seq Num we are waiting for

Xuint8 Ack_Received;        // turns to one in case we got an ACK Message

Xuint32 NbFragRecv;         // contains the previous number of fragments as part of a same message

Xuint32 MsgLength;

Xuint8 ExpectMore;

Xuint32 FragLength;

}CommunicationsVariables;


CommunicationsVariables CommVars[NUMBER_OF_BOARDS];
                                // holds 5 sets of variables for comm between board 0 to 4 in MAC terms

                                // MAC Addresses are :
```

```
//
// 0  --> 06 05 04 03 02 00
// 1  --> 06 05 04 03 02 01
// 2  --> 06 05 04 03 02 02
// 3  --> 06 05 04 03 02 03
// 4  --> 06 05 04 03 02 04
//

#endif
```

mb0_HandlerFunctions.h

```
/*****************************************************************************
*
*       Simple Reliable Communication System over Ethernet _ Microblaze 0
*
*       This code contains the definitions of various interrupt handler functions
*
*       Author:         Patrick E. Akl
*               American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher-John Comis (University of Toronto)
*   at the following email address:
*                                   comis@eecg.toronto.edu
*
```

```
******************************************************************************/

#ifndef MB0_HANDLERFUNCTIONS_H
#define MB0_HANDLERFUNCTIONS_H

#include "mb0_GlobalVariables.h"
#include "mb0_DebugFunctions.h"
#include "mb0_SendFunctions.h"

/*
* This function takes care of the incoming messages and writes payload into
* the receive buffers defined in GlobalVariables.h
*
* It also calls the send functions to send Acknowledge frames when
* necessary
*/

static void FifoRecvHandler(void *CallBackRef);
/*
* Sets Timer_Expired to 1 when timer counter expires
*
* This value is used in the SendReliably function as an indicator
* that we did not receive an ACK in the acceptable delay
*
* It is disabled not to generate new interrupts.
* We enable it only when we start counting after a send.
*
* CallBackRef is a pointer to the XTmrCtr
*/
```

```
static void TimerCounterHandler(void *CallBackRef);

/*
* Checks for errors.
*/
static void FifoSendHandler(void *CallBackRef);

/*
* Called in case of Errors to reset the system
*/
static void ErrorHandler(void *CallBackRef, XStatus Code);


#endif
```

mb0_HandlerFunctions.c

```
/*****************************************************************************
*
*       Simple Reliable Communication System over Ethernet _ Microblaze 0
*
*       This code contains the source code of various interrupt handler functions
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
```

```
*
*       or contact Christopher-John Comis (University of Toronto)
*       at the following email address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/

#include "mb0_GlobalVariables.h"
#include "mb0_DebugFunctions.h"
#include "mb0_SendFunctions.h"
#include "mb0_HandlerFunctions.h"

/*
* This function takes care of the incoming messages and writes payload into
* the receive buffers defined in GlobalVariables.h
*
* It also calls the send functions to send Acknowledge frames when
* necessary
*/
static void FifoRecvHandler(void *CallBackRef)
{
        Xuint32 CommVarsIndex;
        XStatus Result;
        Xuint32 FrameLen;
        Xuint32 writingcounter;
        Xuint8* TempAddressPtr;
        Xuint8 * ReadingPtr;
        Xuint32 MessageLen;
        Xuint32 read_v;
```

```
	XIntc_Disable(&InterruptController, INTC_INPUT_EMAC);

/*Receive the Frame*/
	FrameLen = XEM_MAX_FRAME_SIZE;
	Result = XEmac_FifoRecv((XEmac *)CallBackRef, RecvBuffer, &FrameLen);
	MessageLen = GetSize(RecvBuffer);

/*Check for validity*/
	if (Result != XST_SUCCESS)
	{
		xil_printf("Fifo Receive Error\n\r");
		XIntc_Enable(&InterruptController, INTC_INPUT_EMAC);

		return;
	}   // end of /*Check for validity*/


/*Valid Frame : ACK or Message*/
	else
	{
		TempAddressPtr = (Xuint8 *)RecvBuffer + XEM_MAC_ADDR_SIZE;
		CommVarsIndex = GetIndex((Xuint8 *)TempAddressPtr);

/*ACK with valid SEQNUM*/
		if(GetType((Xuint8 *)RecvBuffer) == ACK_MESSAGE_TYPE  &&  GetSeqNum((Xuint8 *)RecvBuffer) ==
CommVars[CommVarsIndex].SeqNum)
		{
				XTmrCtr_Stop(&TimerCounter, TIMER_COUNTER_0);
				XIntc_Disable(&InterruptController, INTC_INPUT_TIMER);
```

```
                        CommVars[CommVarsIndex].Ack_Received =  1;
                }       // end of /*ACK with valid SEQNUM*/


/*Message*/
                if(GetType((Xuint8 *)RecvBuffer) != ACK_MESSAGE_TYPE)
                {
                        if(GetSeqNum((Xuint8 *)RecvBuffer) == CommVars[CommVarsIndex].WaitingFrameNumber - 1)
                        {
                                Send_Ack_Message(RecvBuffer);
                        }


                        if(GetSeqNum((Xuint8 *)RecvBuffer) == CommVars[CommVarsIndex].WaitingFrameNumber)
                        {
                                if(BusArbitration == 0) // we have a trial
                                {
                                        Interrupt_Second_MicroBlaze();

                                        if (BusArbitration == 0)
                                        {
        /*Valid Message*/

                                                if(GetType((Xuint8 *)RecvBuffer) == FIRST_FRAGMENT_TYPE)
                                                {
        /*First Fragment Type*/
                                                        CommVars[CommVarsIndex].ExpectMore = 1;
                                                        CommVars[CommVarsIndex].NbFragRecv = 1;

                                                        ReadingPtr = (Xuint8 *)RecvBuffer;
```

```
                                        ReadingPtr += 24;

                                        TempAddressPtr = (Xuint8 *)RecvBuffer + XEM_MAC_ADDR_SIZE;

                                        Send_Frame_to_App(ReadingPtr, MessageLen, 1490, TempAddressPtr,
FIRST_FRAGMENT_TYPE);
                                        microblaze_bread_datafsl(read_v, 1);
                                        Send_Ack_Message(RecvBuffer);   // wait for the user to read the frame and then send an ACK


                                }

        /*Increment the waiting sequence number*/
                                        CommVars[CommVarsIndex].WaitingFrameNumber = (
CommVars[CommVarsIndex].WaitingFrameNumber + 1 ) % 0xFFFFFFFF;

                                        if(GetType((Xuint8 *)RecvBuffer) == NON_FRAGMENTED_MESSAGE_TYPE)
                                        {
        /*Non Fragmented Message Frame Type*/
                                        CommVars[CommVarsIndex].ExpectMore = 0;
                                        CommVars[CommVarsIndex].NbFragRecv = 1;

                                        ReadingPtr = (Xuint8 *)RecvBuffer;
                                        ReadingPtr += 24;

                                        CommVars[CommVarsIndex].MsgLength = GetSize(&RecvBuffer);

                                        TempAddressPtr = (Xuint8 *)RecvBuffer + XEM_MAC_ADDR_SIZE;
```

```
                    Send_Frame_to_App(ReadingPtr, MessageLen, MessageLen, TempAddressPtr,
NON_FRAGMENTED_MESSAGE_TYPE);
                    microblaze_bread_datafsl(read_v, 1);
                    Send_Ack_Message(RecvBuffer);   // wait for the user to read the frame and then send an ACK


                }       // end of /*Non Fragmented Message Frame Type*/

                if(GetType((Xuint8 *)RecvBuffer) == INTERNAL_FRAGMENT_TYPE)
                {
        /*Internal Fragment Type*/
                    CommVars[CommVarsIndex].NbFragRecv ++;
                    CommVars[CommVarsIndex].ExpectMore = 1;

                    ReadingPtr = (Xuint8 *)RecvBuffer;
                    ReadingPtr += 24;

                    TempAddressPtr = (Xuint8 *)RecvBuffer + XEM_MAC_ADDR_SIZE;

                    Send_Frame_to_App(ReadingPtr, MessageLen, 1490, TempAddressPtr,
INTERNAL_FRAGMENT_TYPE);
                    microblaze_bread_datafsl(read_v, 1);
                    Send_Ack_Message(RecvBuffer);   // wait for the user to read the frame and then send an ACK


                } // end of /*Internal Fragment Type*/

                if(GetType((Xuint8 *)RecvBuffer) == LAST_FRAGMENT_TYPE)
                {
        /*Last Fragment Type*/
```

```
                                        CommVars[CommVarsIndex].NbFragRecv++;
                                        CommVars[CommVarsIndex].ExpectMore = 0;

                                        CommVars[CommVarsIndex].MsgLength = GetSize(&RecvBuffer);

                                        if(CommVars[CommVarsIndex].MsgLength % MAX_L2_PAYLOAD == 0)
                                                CommVars[CommVarsIndex].FragLength = MAX_L2_PAYLOAD;
                                        else
                                                CommVars[CommVarsIndex].FragLength = CommVars[CommVarsIndex].MsgLength %
MAX_L2_PAYLOAD;

                                        ReadingPtr = (Xuint8 *)RecvBuffer;
                                        ReadingPtr += 24;

                                        TempAddressPtr = (Xuint8 *)RecvBuffer + XEM_MAC_ADDR_SIZE;

                                        Send_Frame_to_App(ReadingPtr, MessageLen, CommVars[CommVarsIndex].FragLength,
TempAddressPtr, LAST_FRAGMENT_TYPE);
                                        microblaze_bread_datafsl(read_v, 1);
                                        Send_Ack_Message(RecvBuffer);   // wait for the user to read the frame and then send an ACK


                                                }       // end of /*Last Fragment Type*/
                                        }       // end of BusArbitration == 0 after the Interruption
                                } // end of BusArbitration == 0
                        }       // end of Valid Seqnum
                }   // end of /* Message */
        } // end of /*Valid Frame : ACK or Message*/

        XIntc_Enable(&InterruptController, INTC_INPUT_EMAC);
```

```
}

/*
* Sets Timer_Expired to 1 when timer counter expires
*
* This value is used in the SendReliably function as an indicator
* that we did not receive an ACK in the acceptable delay
*
* It is disabled not to generate new interrupts.
* We enable it only when we start counting after a send.
*
* CallBackRef is a pointer to the XTmrCtr
*/

static void TimerCounterHandler(void *CallBackRef)
{
        Timer_Expired = 1;
   xil_printf("ACK Time Exp.\n\r");
        XIntc_Disable(&InterruptController, INTC_INPUT_TIMER);
}

/*
* Checks for errors.
*/

static void FifoSendHandler(void *CallBackRef)
{
   XEmac_GetStats((XEmac *)CallBackRef, &Stats);

        if (Stats.XmitLateCollisionErrors || Stats.XmitExcessDeferral)
```

```
   {
                xil_printf("Late Collision Number : %d\n\r", Stats.XmitLateCollisionErrors);
                xil_printf("Late Deferrals Number : %d\n\r", Stats.XmitExcessDeferral);
                xil_printf("Error in FIFO Sending !!\n\r");
                PrintEmacStats(Stats);
   }
        XEmac_ClearStats((XEmac *)CallBackRef);
}

/*
* Called in case of Errors to reset the system
*/
static void ErrorHandler(void *CallBackRef, XStatus Code)
{
        xil_printf("Starting FIFO Error Handler\n\r");
   if (Code == XST_RESET_ERROR)
   {
      XEmac_Reset((XEmac *)CallBackRef);                                        // Reset EMAC

      (void)XEmac_SetMacAddress((XEmac *)CallBackRef, LocalAddress);        // SET MAC ADDRESS
      (void)XEmac_SetOptions((XEmac *)CallBackRef, TEST_1_OPTIONS);         // SET OPTIONS

      (void)XEmac_Start((XEmac *)CallBackRef);                                  // Start EMAC
   }
        xil_printf("Exiting FIFO Error Handler\n\r");
}
```

mb0_InitializationFunctions.h

```
/*****************************************************************************
*
*       Simple Reliable Communication System over Ethernet _ Microblaze 0
*
*       This code contains definitions of the system initialization functions
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher-John Comis (University of Toronto)
*       at the following email address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/


#ifndef MB0_INITIALIZATIONFUNCTIONS_H
#define MB0_INITIALIZATIONFUNCTIONS_H

#include "mb0_GlobalVariables.h"
#include "mb0_HandlerFunctions.h"
```

```
/*
* Initializes the GPio
* Led is an output
*/
static XStatus Initialize_GPio();

/*
* Initializes the ACK Frame common fields
* so that we don't do it every time we send an ACK
* at run time.
*
* This is to reduce the computational overhead
* The Ack Frame is contained in the Buffer ACKBuffer[60]
*
* ACK Format:
*
*   DST ADDRESS_SRC ADDRESS_ETHERNET LEN/TYPE FIELD_MESSAGE TYPE_SEQUENCE NUMBERS_MESSAGE LEN_PAYLOAD
*    unknown     automatic          known                    known          unknown                    known    known
*/

/*
*         Initializes the Timercounter with ID 1
*         This Timer Counter is used to interrupt microblaze_1
*
*         Future Work: Using a Timer counter for that purpose is inefficient
*         but I did not find a way to do it otherwise
*         If you look into the add/edit cores menu, you see that the interrupt
*         output timer_intr_1 is connected to the interrupt controller of
*         microblaze_1 input.    Try not to use a timer counter for this.
```

```
*/
void Initialize_MicroBlaze_Interrupt_Timer();


static void InitializeACKFrameCommonFields();

/*
* Setup Interrupt System
*
* EMAC has highest priority                0
* TIMER COUNTER has lowest priority 1
*
* Take a look at the Add / Edit Cores for further details on
* Signal Connections
*/
static XStatus SetupInterruptSystem(XEmac *EmacPtr);

/*
* Initialize the Timer Counter
* This Timer counter is used to retransmit frames after a delay
* in case we got no ack
*/
static XStatus InitializeTimerCounter();

/*
* Initialize the EMAC Module
* for Setup configuration, take a look at TEST_1_OPTIONS variable
* in the GlobalVariables.h Header
*/
static XStatus InitializeEmac();
```

```
/*
* This function is called from Initialize Parameters (which is the only one called from main)
* and hence there is no need to
* call it from main.
*
* It initializes the Communication Parameters Structure contents
*
* It assumes we have 5 boards (check the NUMBER_OF_BOARDS variable in Global Variables)
*/
void Initialize_CommVars();


/*
* This function should be called from main
* It calls all the other functions.
*/
void Initialize_Parameters();

#endif
```

mb0_InitializationFunctions.c

```
/*****************************************************************************
*
*       Simple Reliable Communication System over Ethernet _ Microblaze 0
*
*       This code contains source code of the system initialization functions
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher-John Comis (University of Toronto)
*       at the following email address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/


#include "mb0_GlobalVariables.h"
#include "mb0_HandlerFunctions.h"
#include "mb0_InitializationFunctions.h"

/*
* Initializes the GPio
* Led is an output
```

```
*/
static XStatus Initialize_GPio()
{
        XStatus Status;
   Status = XGpio_Initialize(&Gpio, XPAR_OPB_GPIO_0_DEVICE_ID);
   if (Status != XST_SUCCESS)
   {
      xil_printf("GPIO initialization error !!\n\r");
   }
        else
                xil_printf("GPIO initialization succeeded !!\n\r");
   /* Set the direction for all signals to be inputs except the LED
    * outputs */
   XGpio_SetDataDirection(&Gpio, 1, ~LED);

        return Status;
}

/*
*       Initializes the Timercounter with ID 1
*       This Timer Counter is used to interrupt microblaze_1
*
*       Future Work: Using a Timer counter for that purpose is inefficient
*       but I did not find a way to do it otherwise
*       If you look into the add/edit cores menu, you see that the interrupt
*       output timer_intr_1 is connected to the interrupt controller of
*       microblaze_1 input.    Try not to use a timer counter for this.
*/
void Initialize_MicroBlaze_Interrupt_Timer()
{
```

```
        XTmrCtr_mDisable(0x80010700, 0);
    XTmrCtr_Initialize(&TimerCounter2, 1);
    XTmrCtr_SetHandler(&TimerCounter2, TimerCounterHandler2, &TimerCounter2);
    XTmrCtr_SetOptions(&TimerCounter2, 0, XTC_INT_MODE_OPTION | XTC_AUTO_RELOAD_OPTION);
        XTmrCtr_SetResetValue(&TimerCounter2, 0, 0xFFFFFFFE); // to interrupt immediately
}


/*
* Initializes the ACK Frame common fields
* so that we don't do it every time we send an ACK
* at run time.
*
* This is to reduce the computational overhead
* The Ack Frame is contained in the Buffer ACKBuffer[60]
*
* ACK Format:
*
*   DST ADDRESS_SRC ADDRESS_ETHERNET LEN/TYPE FIELD_MESSAGE TYPE_SEQUENCE NUMBERS_MESSAGE LEN_PAYLOAD
*    unknown     automatic          known                    known          unknown                  known    known
*/

static void InitializeACKFrameCommonFields()
{
        Xuint32 loopcounter;
        Xuint8* ACKBuffPointer;
        ACKBuffPointer = (Xuint8 *) AckFrame;

        // Get Passed Destination and Source MAC Address
        ACKBuffPointer += (2 * XEM_MAC_ADDR_SIZE);
```

```
        // Write Known ETHERNET LEN / TYPE Field _ 46
        *(Xuint16 *)ACKBuffPointer = ACK_L1_PAYLOAD_SIZE;

        // Advance the pointer to the position of Layer 2 Header
        ACKBuffPointer += 2;

        // Write Known Message Type _ 1
        *(Xuint16 *)ACKBuffPointer = ACK_MESSAGE_TYPE;

        // Get Passed SeqNum Field _ Get to Layer2 Len Field
        ACKBuffPointer += 6;

        // Write Known Layer 2 Payload Length Field
        *(Xuint32 *)ACKBuffPointer = ACK_L2_PAYLOAD_SIZE;

        // Advance Pointer to the Layer2 Known ACK Payload
        ACKBuffPointer += 4;

        //Writing Known Layer2 ACK Payload
        for(loopcounter = 0 ; loopcounter < ACK_L2_PAYLOAD_SIZE ; loopcounter++)
        {
                *(Xuint8 *)ACKBuffPointer++ = 0xFF;
        }
}



/*
* Setup Interrupt System
```

```
*
* EMAC has highest priority                    0
* TIMER COUNTER has lowest priority 1
*
* Take a look at the Add / Edit Cores for further details on
* Signal Connections
*/
static XStatus SetupInterruptSystem(XEmac *EmacPtr)                    /* Setup Interrupt System */
{
   XStatus Result;

   microblaze_enable_interrupts();

   Result = XIntc_Initialize(&InterruptController, INTC_DEVICE_ID); // 0 is DEVICE ID for the Interrupt Controller
         if (Result != XST_SUCCESS)
   {
      xil_printf("Error in Xintc Initialization !!\n\r");
                  return Result;
   }
         else
      xil_printf("Xintc Initialization Succeeded !!\n\r");



         Result = XIntc_Connect(&InterruptController, INTC_INPUT_TIMER, (XInterruptHandler)XTmrCtr_InterruptHandler, &TimerCounter);
   if (Result != XST_SUCCESS)
   {
      xil_printf("Error in Xintc Connect to Timer !!\n\r");
                  return Result;
   }
```

```
    else
   xil_printf("Xintc Connect Succeeded to Timer!!\n\r");


    Result = XIntc_Connect(&InterruptController, INTC_INPUT_EMAC, (XInterruptHandler)XEmac_IntrHandlerFifo, EmacPtr);
if (Result != XST_SUCCESS)
{
        return Result;
   xil_printf("Error in Xintc Connect to EMAC!!\n\r");
}
    else
   xil_printf("Xintc Connect Succeeded to EMAC!!\n\r");
                                                    // all interrupts will be serviced in order of occurence


    Result = XIntc_SetOptions(&InterruptController, XIN_SVC_ALL_ISRS_OPTION);
    if (Result != XST_SUCCESS)
{
        return Result;
   xil_printf("Error in Xintc SetOptions !!\n\r");
}
    else
   xil_printf("Xintc SetOptions Succeeded !!\n\r");


    Result = XIntc_Start(&InterruptController, XIN_REAL_MODE);
    if (Result != XST_SUCCESS)
{
        return Result;
   xil_printf("Error in Xintc Start !!\n\r");
```

```
    }
        else
       xil_printf("Xintc Start Succeeded !!\n\r");


        XIntc_Enable(&InterruptController, INTC_INPUT_EMAC); // enable the EMAC interrupt
        //INTC_INPUT_TIMER is enabled in the SendReliably function after a send

    return XST_SUCCESS;

}

/*
* Initialize the Timer Counter
* This Timer counter is used to retransmit frames after a delay
* in case we got no ack
*/
static XStatus InitializeTimerCounter()
{
        XStatus Status;
        // disabling the counter if some are running in the system with low level functions
        XTmrCtr_mDisable(0x80010400, TIMER_COUNTER_0);   // Low Level Function

        // here we disable the intr., the input of the interrupt controller
        XIntc_mMasterDisable(0x80010500);  // Low Level Function

    /************************** Timer Setup **********************************/

    Status = XTmrCtr_Initialize(&TimerCounter, TIMER_COUNTER_0);
                                                    // 0 is the DEVICE ID of the Timer module
```

```
    if (Status != XST_SUCCESS)
    {
        xil_printf("Timer counter initialization error\n\r\r");
                return Status;
    }

    Status = XTmrCtr_SelfTest(&TimerCounter, TIMER_COUNTER_0);
    if (Status != XST_SUCCESS)
    {
        xil_printf("Timer counter self-test error\n\r");
                return Status;

    }

    XTmrCtr_SetHandler(&TimerCounter, TimerCounterHandler, &TimerCounter);

    XTmrCtr_SetOptions(&TimerCounter, TIMER_COUNTER_0, XTC_INT_MODE_OPTION | XTC_AUTO_RELOAD_OPTION);

    XTmrCtr_SetResetValue(&TimerCounter, TIMER_COUNTER_0, 0xF0000000);

    return XST_SUCCESS;
}

/*
* Initialize the EMAC Module
* for Setup configuration, take a look at TEST_1_OPTIONS variable
* in the GlobalVariables.h Header
*/
static XStatus InitializeEmac()
{
```

```
	XEmac_Config *ConfigPtr;
	XStatus Result;
	Xuint32 DeviceId;
	DeviceId = 0;
	EmacPtr = & Emac;

ConfigPtr = XEmac_LookupConfig(DeviceId);
ConfigPtr->IpIfDmaConfig = XEM_CFG_NO_DMA;

Result = XEmac_Initialize(EmacPtr, DeviceId);
if (Result != XST_SUCCESS)
{
		xil_printf("Error in EMAC Initialization !!\n\r");
   return Result;
}
	else
		xil_printf("Initialization Succeeded !!\n\r");


Result = XEmac_SelfTest(EmacPtr);
if (Result != XST_SUCCESS)
{
		xil_printf("Self Test Error !!\n\r");
   return Result;
}
	else
		xil_printf("Self Test Succeeded !!\n\r");

Result = XEmac_SetOptions(EmacPtr, TEST_1_OPTIONS);
if (Result != XST_SUCCESS)
```

```
{
		xil_printf("Set Options Error !!\n\r");
   return Result;
}
	else
		xil_printf("Set Options Succeeded !!\n\r");

Result = XEmac_SetMacAddress(EmacPtr, LocalAddress);
if (Result != XST_SUCCESS)
{
		xil_printf("Set MAC Address Error !!\n\r");
   return Result;
}
	else
		xil_printf("Set MAC Address Succeeded !!\n\r");

XEmac_SetFifoSendHandler(EmacPtr, EmacPtr, FifoSendHandler);
XEmac_SetFifoRecvHandler(EmacPtr, EmacPtr, FifoRecvHandler);
XEmac_SetErrorHandler(EmacPtr, EmacPtr, ErrorHandler);


	XEmac_ClearStats(EmacPtr);
Result = XEmac_Start(EmacPtr);
if (Result != XST_SUCCESS)
{
		xil_printf("EMAC Start Error !!\n\r");
   return Result;
}
	else
		xil_printf("EMAC Start Success !!\n\r");
```

```
	return XST_SUCCESS;
}



/*
* This function is called from Initialize Parameters (which is the only one called from main)
* and hence there is no need to
* call it from main.
*
* It initializes the Communication Parameters Structure contents
*
* It assumes we have 5 boards (check the NUMBER_OF_BOARDS variable in Global Variables)
*/
void Initialize_CommVars()
{
	Xuint8 cntr;

	for(cntr=0 ; cntr < NUMBER_OF_BOARDS ; cntr++)
	{

		CommVars[cntr].SeqNum = 0;
		CommVars[cntr].WaitingFrameNumber = 0;
		CommVars[cntr].Ack_Received = 0;
		CommVars[cntr].ExpectMore = 0;
		CommVars[cntr].NbFragRecv = 0;
		CommVars[cntr].MsgLength = 0;
		CommVars[cntr].FragLength = 0;
```

```
        }
}


/*
* This function should be called from main
* It calls all the other functions.
*/
void Initialize_Parameters()
{
        XStatus Result;

        Initialize_CommVars();

        Initialize_MicroBlaze_Interrupt_Timer();

        InitializeACKFrameCommonFields();

        Result = InitializeEmac();
        if (Result != XST_SUCCESS)
   {
                xil_printf("EMAC Initialization Error !!\n\r");
      return;
   }
        else
                xil_printf("EMAC Initialization Success !!\n\r");


        Result = InitializeTimerCounter();
        if (Result != XST_SUCCESS)
```

```
    {
                xil_printf("TimerCounter Initialization Error !!\n\r");
       return;
    }
        else
                xil_printf("TimerCounter Initialization Success !!\n\r");

    Result = SetupInterruptSystem(EmacPtr);
    if (Result != XST_SUCCESS)
    {
                xil_printf("Set Interrupt System Error !!\n\r");
       return;
    }
        else
                xil_printf("Set Interrupt System Success !!\n\r");


        Result = Initialize_GPio();
    if (Result != XST_SUCCESS)
    {
                xil_printf("GPio Initialization Error !!\n\r");
       return;
    }
        else
                xil_printf("GPio Initialization Success !!\n\r");
}
```

mb0_SendFunctions.h

```
/*****************************************************************************
*
*       Simple Reliable Communication System over Ethernet _ Microblaze 0
*
*       This code contains definitions of the send functions of the various network layers
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher-John Comis (University of Toronto)
*       at the following email address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/

#ifndef MB0_SENDFUNCTIONS_H
#define MB0_SENDFUNCTIONS_H

#include "mb0_GlobalVariables.h"
#include "mb0_DebugFunctions.h"

/*
* Fills the Ethernet Header and the Payload into TxFrame Buffer
* Source Address automatically included by PHY
```

```
*
* returns XST_SUCCESS or XST_FAILURE for just sending the frame
*
* Further reliability should be implemented seperatly
*
* Ethernet Header Format:     Destination Address _ 6 bytes
*                                                 Source Address      _ 6 bytes
*                         Len / Type Field    _ 2 bytes
*/
static XStatus SendFrame(XEmac *EmacPtr, Xuint32 FramePayloadSize, Xuint8* InputBuffer,
                         Xuint8 *DestAddress);

/*
* Pass it a pointer to the message we want to send an ACK for
*
* Destination MAC address for the ACK determined automatically
*
* NOTE: this will send an ACK only for frames that are non ACK Messages
*       The check for this is in the 'if' statement
*
* Returns XST_SUCCESS or XST_FAILURE
*
* XST_FAILURE      if we failed to send
*                                 or if we tried to send an ACK for an ACK
*/
XStatus Send_Ack_Message(Xuint8 *MsgBuffer);
/*
* Fills the Layer 2 Header and the Payload into TxMessage Buffer
* It calls the SendFrame to fill the Ethernet header and send the frame
*
```

```
* Note: the messages sent by this function can only be <= 1490 bytes in length
* returns XST_SUCCESS or XST_FAILURE for just sending the frame
*
* Further reliability should be implemented seperatly
*
*  Message Header Format:    Type                    _ 2 bytes
*                                                    Sequence Number       _ 4 bytes
*                       Message Length      _ 4 bytes
*/
static XStatus SendMessage(Xuint32 MessagePayloadSz, Xuint8* InputBuffer,
                    Xuint8 *DestAddress,  Xuint16 TypeofMessage);

/*
* Uses send message to send the payload passed through the pointer InputBuffer
*
* tried to send it 10 times after which it returns XST_FAILURE
* returns XST_SUCCESS if an ACK is received for this message
*
* Uses different waiting times for resetting the timer.
* Once the timer expires it sets Timer_Expired = 1
*
* Ack_Received is set to 1 by the receive handler function
*
*
* Type of Message:
* 0 Non Fragmented Message
* 1 ACK Message
* 2 First Fragment
* 3 Internal Fragment
* 4 Last Fragment
```

```
*/
static XStatus SendReliably(Xuint32 MessagePayloadSz, Xuint8 *InputBuffer, Xuint8 *DestAddress, Xuint16 TypeofMessage);

#endif
```

mb0_SendFunctions.c

```
/*****************************************************************************
*
*       Simple Reliable Communication System over Ethernet _ Microblaze 0
*
*       This code contains source code of the send functions of the various network layers
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher-John Comis (University of Toronto)
*       at the following email address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/


#include "mb0_GlobalVariables.h"
#include "mb0_DebugFunctions.h"
```

```
#include "mb0_SendFunctions.h"

/*
* Fills the Ethernet Header and the Payload into TxFrame Buffer
* Source Address automatically included by PHY
*
* returns XST_SUCCESS or XST_FAILURE for just sending the frame
*
* Further reliability should be implemented seperatly
*
* Ethernet Header Format:     Destination Address _ 6 bytes
*                                                 Source Address      _ 6 bytes
*                       Len / Type Field    _ 2 bytes
*/
static XStatus SendFrame(XEmac *EmacPtr, Xuint32 FramePayloadSize, Xuint8* InputBuffer,
                    Xuint8 *DestAddress)
{
        Xuint32 Index;
   Xuint8 *FramePtr;
   Xuint8 *AddrPtr = DestAddress;

   FramePtr = (Xuint8 *)TxFrame;

   *FramePtr++ = *AddrPtr++;                    // Writing destination address
   *FramePtr++ = *AddrPtr++;
   *FramePtr++ = *AddrPtr++;
   *FramePtr++ = *AddrPtr++;
   *FramePtr++ = *AddrPtr++;
   *FramePtr++ = *AddrPtr++;
                                                                  // Source address automatically written
```

```
                                                        // just keep place for it
    FramePtr += XEM_MAC_ADDR_SIZE;

                                                        // Write payload Size
    *(Xuint16 *)FramePtr = (Xuint16)FramePayloadSize;

          FramePtr += 2;
                        // Write Payload
          for (Index = 0; Index < FramePayloadSize ; Index++)
    {
       *FramePtr++ = *InputBuffer++;
    }

    return XEmac_FifoSend(EmacPtr, (Xuint8 *)TxFrame, FramePayloadSize + XEM_HDR_SIZE);
}



/*
* Pass it a pointer to the message we want to send an ACK for
*
* Destination MAC address for the ACK determined automatically
*
* NOTE: this will send an ACK only for frames that are non ACK Messages
*       The check for this is in the 'if' statement
*
* Returns XST_SUCCESS or XST_FAILURE
*
* XST_FAILURE      if we failed to send
*                                  or if we tried to send an ACK for an ACK
```

```
*/
XStatus Send_Ack_Message(Xuint8 *MsgBuffer)
{
        Xuint32 mycounter;
   Xuint8 *MessagePtr;
        Xuint8 *SourceAdd;                              // will hold the ack destination address
                                                                // which is the received message source address
        Xuint32 RecvSeqNumber;                          // Received Seq. Num will be sent back in the ACK

        Xuint8 *MesgPtr = MsgBuffer;

        MesgPtr+=6;                                     // Get destination address Pointer
        SourceAdd = MesgPtr;

        MesgPtr+=8;

        if(*(Xuint16 *)MesgPtr!= ACK_MESSAGE_TYPE)                      // Don't send an ACK for an ACK !!
        {
                MesgPtr+=2;

                RecvSeqNumber = *(Xuint32 *)MesgPtr;// Get the sequence number

                // Now Write the Specific Fields Frame To Send in Pre Written ACK Frame

                MessagePtr = (Xuint8 *)AckFrame;  // AckFrame will contain the whole ACK frame

                *MessagePtr++ = *SourceAdd++;           // Copying the Destination Address
                *MessagePtr++ = *SourceAdd++;
                *MessagePtr++ = *SourceAdd++;
                *MessagePtr++ = *SourceAdd++;
```

```
                *MessagePtr++ = *SourceAdd++;
                *MessagePtr++ = *SourceAdd++;

                // Get Past Local Address (Automatically Written)   _6 bytes
                // Get Past Ethernet Len Field                                  _2 bytes
                // Get Past Layer 2 Type Field                                          _2 bytes
                // To get to the Sequence Number Position
                MessagePtr += 10;

                // Writing Sequence Number
                *(Xuint32*)MessagePtr = RecvSeqNumber; // Sequence Number

                return XEmac_FifoSend(EmacPtr, (Xuint8 *)AckFrame, ACK_FRAME_SIZE);
        }
        return XST_FAILURE; // we tried to send an ACK for an ACK
}


/*
* Fills the Layer 2 Header and the Payload into TxMessage Buffer
* It calls the SendFrame to fill the Ethernet header and send the frame
*
* Note: the messages sent by this function can only be <= 1490 bytes in length
* returns XST_SUCCESS or XST_FAILURE for just sending the frame
*
* Further reliability should be implemented seperatly
*
*  Message Header Format:    Type                    _ 2 bytes
*                                                    Sequence Number       _ 4 bytes
*                       Message Length      _ 4 bytes
```

```
*/
static XStatus SendMessage(Xuint32 MessagePayloadSz, Xuint8* InputBuffer,
                    Xuint8 *DestAddress,  Xuint16 TypeofMessage)
{
        Xuint32 CommVarsIndex;
        Xuint32 mycounter;
   Xuint8 *TemporaryPtr;
   Xuint8 *MessagePtr;
        Xuint32 FrPaySz;
        Xuint8*TempAddressPtr = DestAddress;

        CommVarsIndex = GetIndex((Xuint8 *)TempAddressPtr);

        TemporaryPtr = TemporaryBuffer;
   MessagePtr = (Xuint8 *)TxMessage;

        *(Xuint16*)MessagePtr = TypeofMessage;              // Type of Message

        MessagePtr+=2;

        *(Xuint32*)TemporaryPtr = CommVars[CommVarsIndex].SeqNum;                       // Sequence Number

        *(Xuint16*)MessagePtr = *(Xuint16 *)TemporaryPtr;

        MessagePtr+=2;
        TemporaryPtr+=2;

        *(Xuint16*)MessagePtr = *(Xuint16 *)TemporaryPtr;

        MessagePtr+=2;
```

```
TemporaryPtr+=2;

TemporaryPtr = TemporaryBuffer;
*(Xuint32*)TemporaryPtr = MessagePayloadSz;   // Length of Message

*(Xuint16*)MessagePtr = *(Xuint16 *)TemporaryPtr;
MessagePtr+=2;
TemporaryPtr+=2;

*(Xuint16*)MessagePtr = *(Xuint16 *)TemporaryPtr;
MessagePtr+=2;
TemporaryPtr+=2;

TemporaryPtr = InputBuffer;

if(MessagePayloadSz <=MAX_L2_PAYLOAD)
        FrPaySz = MessagePayloadSz + L2_HEADER_SIZE;        // The Frame contains the Message header (10 bytes)
                                                                // and the whole message itself as its payload
  // case of fragmented message
else
{
        if(TypeofMessage == FIRST_FRAGMENT_TYPE || TypeofMessage == INTERNAL_FRAGMENT_TYPE) // frame is full
                FrPaySz = L2_HEADER_SIZE + MAX_L2_PAYLOAD;  // 1490 + 10
        else if (TypeofMessage == LAST_FRAGMENT_TYPE) // last fragment of a message
        {
                if(MessagePayloadSz % MAX_L2_PAYLOAD == 0) // 1490 payload size of last fragment
                        FrPaySz = L2_HEADER_SIZE + MAX_L2_PAYLOAD; // 1490 + 10
                else
                        FrPaySz = L2_HEADER_SIZE + (MessagePayloadSz % MAX_L2_PAYLOAD);
        }
```

```
        }

        for(mycounter = 0 ; mycounter < ( FrPaySz - L2_HEADER_SIZE ) ; mycounter++)
        {
                *MessagePtr++ = *TemporaryPtr++;  // fill layer 2 payload
        }

        return SendFrame(EmacPtr, FrPaySz,  TxMessage, DestAddress);
}


/*
* Uses send message to send the payload passed through the pointer InputBuffer
*
* tried to send it 10 times after which it returns XST_FAILURE
* returns XST_SUCCESS if an ACK is received for this message
*
* Uses different waiting times for resetting the timer.
* Once the timer expires it sets Timer_Expired = 1
*
* Ack_Received is set to 1 by the receive handler function
*
*
* Type of Message:
* 0 Non Fragmented Message
* 1 ACK Message
* 2 First Fragment
* 3 Internal Fragment
* 4 Last Fragment
*/
```

```
static XStatus SendReliably(Xuint32 MessagePayloadSz, Xuint8 *InputBuffer, Xuint8 *DestAddress, Xuint16 TypeofMessage)
{
	Xuint32 CommVarsIndex;
	Xuint8* TempAddressPtr;
	Xuint8 *MssPtr;
	Xuint8 RetransmissionCounter=0;	// we still did not make retransmissions

	TempAddressPtr = DestAddress;
	CommVarsIndex = GetIndex((Xuint8 *)TempAddressPtr);


	MssPtr = InputBuffer;
	CommVars[CommVarsIndex].Ack_Received = 0;
	Timer_Expired = 1;					// To enter the first time the while loop

	while(CommVars[CommVarsIndex].Ack_Received == 0 && RetransmissionCounter < MAX_RETRANSMIT_ATTEMPTS)
	{
		if(Timer_Expired == 1) // The timer wrapped back or we are in for the first time
		{

			RetransmissionCounter++;
			Timer_Expired = 0;

			if(RetransmissionCounter != MAX_RETRANSMIT_ATTEMPTS)
				SendMessage(MessagePayloadSz, MssPtr, DestAddress, TypeofMessage);

			XTmrCtr_SetResetValue(&TimerCounter,TIMER_COUNTER_0,
RstVal[RetransmissionCounter%MAX_RETRANSMIT_ATTEMPTS]);
					// Setting the reset value
			XIntc_Acknowledge(&InterruptController, INTC_INPUT_TIMER);
```

```
                        // deleting to 'disable' all interrupts previously done by the running counter
                XIntc_Enable(&InterruptController, INTC_INPUT_TIMER);
                        // Now we accept Acknowledgments and Pass them to Microblaze
                XTmrCtr_Start(&TimerCounter, TIMER_COUNTER_0);
                        // Starting to count

            }
        }
        XIntc_Disable(&InterruptController, INTC_INPUT_TIMER);
                        // Don't let the interrupts constantly generated by the counter call the timer handler
        if(RetransmissionCounter == MAX_RETRANSMIT_ATTEMPTS && CommVars[CommVarsIndex].Ack_Received == 0)
            return XST_FAILURE;

        CommVars[CommVarsIndex].SeqNum = ( CommVars[CommVarsIndex].SeqNum + 1 ) % 0xFFFFFFFF;        // in case we are not able to
transmit,
        return XST_SUCCESS;                                             // the sequence number stays the same
}
```

mb1.c

```
/*****************************************************************************
*
*       Reliable Communication System over Ethernet _ Microblaze 1
*
*       This code is provided for the user to write his application that will be running
*       on microblaze_1. Some Functions are provided to send data to microblaze_0 which will
*       handle the reliable transfer over Ethernet to the destination board.
*
*       Note that the data we receive will be transmitted via fsl links from microblaze_0
*       which contains the EMAC Controller to microblaze_1
*       the receive buffer is filled in the Handler function. It is the responsability of the user
*       to read the data into the user defined application buffer. microblaze_0 will pass to
*       microblaze_1 the total message size as well as the source MAC Address for the user
*       to demultiplex the data.
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher John Comis (University of Toronto)
*       at the following address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/
```

```
#include "xparameters.h"
#include "xstatus.h"
#include "mb_interface.h"
#include "xintc.h"

#include "MACAddresses.h"
#include "mb1_GlobalVariables.h"              // contains the global variables used on this microblaze
#include "mb1_Functions.h"      // contains the fsl transfer functions as well as the other
                                                                        // functions used on microblaze_1
#include "MACAddresses.c"
#include "mb1_functions.c"

#include "mb1_FSLTransferFunctions.h"
#include "mb1_HandlerFunctions.h"
#include "mb1_InitializationFunctions.h"
#include "mb1_SendFunctions.h"

#include "mb1_FSLTransferFunctions.c"
#include "mb1_HandlerFunctions.c"
#include "mb1_InitializationFunctions.c"
#include "mb1_SendFunctions.c"



/******************* Main Function (this is where the user code should go) **********/

int main()
{
        /* Variable definitions          */
```

```
Xuint32 val;
Xuint32 counter;
Xuint32 counter2;
XStatus SystemReady;

xil_printf("\n\n\rMicroBlaze 1 Setup\n\r");
xil_printf("------------------\n\n\r");

/* MicroBlaze System Setup */
SystemReady = Initialize_Parameters();

if(SystemReady != XST_SUCCESS)
{
        xil_printf("System Failure -- Exiting Main\n\n\r");
        return 0;       // exit main
}

xil_printf("\n\rMicroBlaze 1 Ready\n\r");
xil_printf("------------------\n\n\r");


// this should be specific to communication with
// every other node

/* Synchronization with the second microblaze       */
/* So that they enter the "main" together           */

microblaze_bread_datafsl(val,0);
microblaze_bwrite_datafsl(10, 1);
```

```
/*****************************************/
/* "Main": this is where your code starts */
/*****************************************/

        /* fills the Buffer used to send the message with known
           payload (a counter % 100), this is just for testing */
Fill_Buffer(10000, Message_To_Send_Buffer);

        /* Delay Loop */

for(counter = 0 ; counter < 10000000 ; counter ++);

        /*      Sending a message of 535 bytes
                pointed to by Message_To_Send_Buffer
                with Address3 as the destination */

if (Send_EMAC(535, Message_To_Send_Buffer, Address3) == 0)
{
        xil_printf("Succeeded in Sending First Message !!\n\r");
}
else // == 1
{
        xil_printf("Failed in Sending First Message !!\n\r");
}


        /*      Introducing Some Delay between the two sends
                for the test. This is just to be able to monitor the
                transfers and see what happens */
for(counter = 0 ; counter < 10000000 ; counter ++);
```

```
        /*      Sending a message of 3500 bytes
                pointed to by Message_To_Send_Buffer
                with Address3 as the destination */
if (Send_EMAC(2345, Message_To_Send_Buffer, Address3) == 0)
{
        xil_printf("Succeeded in Sending Second Message !!\n\r");
}
else // == 1
{
        xil_printf("Failed in Sending Second Message !!\n\r");
}


// NOTE :       For the reception of data from other boards
//                      The user is responsible to write some code in the
//                      Handler function to  demultiplex data into the
//                      appropriate buffer


/*************************************************/
/* end of "Main": This is where your code ends */
/*************************************************/

/* While Loop to keep the program running to be able to receive data        */

while(1)
{

}
```

```
	return 0;
}

/*****************************************************************************/
```

mb1_FSLTransferFunction.h

```
/*****************************************************************************
*
*	Simple Reliable Communication System over Ethernet _ Microblaze 0
*
*	Contains all the functions used by microblaze_1
*
*	Author:		Patrick E. Akl
*			American University of Beirut
*			August 2004
*
*	For any comments, suggestions or bug discovery, please contact me at the following
*	email address : patrickakl@hotmail.com
*
*	or contact Christopher-John Comis (University of Toronto)
*	at the following email address:
*				comis@eecg.toronto.edu
*
*****************************************************************************/
```

```
#ifndef MB1_FSLTRANSFERFUNCTIONS_H
#define MB1_FSLTRANSFERFUNCTIONS_H

#include "mb1_GlobalVariables.h"
#include "mb1_Functions.h"

/*
* Takes a pointer to the control message
* will set the global variables TotalMessageSize
* and DestinationAddress array, and TypeofMessage
*
* by calling fill_array(pointer to the MAC Address location)
* we will fill 'array' with the EMAC printable address in standard format
* Use xil_printf("%s", array)
*/
void Parse_Control_Message(Xuint8 * BufferPointer);



/********************************** Get Functions ****************************/

/*
*       takes a pointer to a buffer where to write and the number of bytes to read from the fsl
*       connected to microblaze_0
*
*       This function will determine the number of fsl word transfers and perform blocking reads
*       on the fsl0 (microblaze_0 is the master on fsl0, and microblaze_1 is the slave)
*/
void Get_from_EMAC(Xuint8 *BuffPtr, Xuint32 Length_Bytes);
```

```
/*
*       This function will read first a control message (MessageTotalSize and Source Address)
*       It uses this data to know what data to expect next
*
*       a Get_from_EMAC is used to read the data payload from the fsls
*
*       Returns the CommVarsIndex for the user to be able to use it
*       it will enable him to know about the sender and about the state of
*       data sender ...
*/
Xuint32 Get_Frame_from_EMAC(Xuint8 *BuffPtr);


#endif
```

mb1_FSLTransferFunction.c

```
/*****************************************************************************
*
*       Reliable Communication System over Ethernet _ Microblaze 1
*
*       This Code contains the source code for the FSL transfer Get Functions
*       These are used to get data from the second microblaze via fsl transfers
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher John Comis (University of Toronto)
*       at the following address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/

#include "mb1_FSLTransferFunctions.h"


/*
* Takes a pointer to the control message
* will set the global variables TotalMessageSize
* and DestinationAddress array
```

```
*
* by calling fill_array(pointer to the MAC Address location)
* we will fill 'array' with the EMAC printable address in standard format
* Use xil_printf("%s", array)
*/
void Parse_Control_Message(Xuint8 * BufferPointer)
{
	Xuint8 * Ptr = BufferPointer;

	TotalMessageSize = *(Xuint32 *)Ptr;

	Ptr+=4;

	fill_array(Ptr);

	DestinationAddress[0] = *Ptr++;
	DestinationAddress[1] = *Ptr++;
	DestinationAddress[2] = *Ptr++;
	DestinationAddress[3] = *Ptr++;
	DestinationAddress[4] = *Ptr++;
	DestinationAddress[5] = *Ptr++;

	TypeofMessage = *(Xuint16 *)Ptr;
}


/*
*	takes a pointer to a buffer where to write and the number of bytes to read from the fsl
*	connected to microblaze_0
*
```

```
*       This function will determine the number of fsl word transfers and perform blocking reads
*       on the fsl0 (microblaze_0 is the master on fsl0, and microblaze_1 is the slave)
*/
void Get_from_EMAC(Xuint8 *BuffPtr, Xuint32 Length_Bytes)
{
        Xuint32 TempSize = Length_Bytes;
        Xuint32 Temp;

        Xuint8 *Ptr = BuffPtr;
        Xuint8 *Pointer;

        while(TempSize >= 4)
        {
                microblaze_bread_datafsl(Temp, 0);

                Pointer = (Xuint8 *)&Temp;

                *Ptr++ = *Pointer++;
                *Ptr++ = *Pointer++;
                *Ptr++ = *Pointer++;
                *Ptr++ = *Pointer++;

                TempSize = TempSize - 4; // read 4 bytes
        }
        if(TempSize == 3)
        {
                microblaze_bread_datafsl(Temp, 0);

                Pointer = (Xuint8 *)&Temp;
```

```
                *Ptr++ = *Pointer++;
                *Ptr++ = *Pointer++;
                *Ptr++ = *Pointer++;
        }

        else if(TempSize == 2)
        {
                microblaze_bread_datafsl(Temp, 0);

                Pointer = (Xuint8 *)&Temp;

                *Ptr++ = *Pointer++;
                *Ptr++ = *Pointer++;
        }
        else if(TempSize == 1)
        {
                microblaze_bread_datafsl(Temp, 0);

                Pointer = (Xuint8 *)&Temp;

                *Ptr++ = *Pointer++;
        }

}



/*
```

```
*       This function will read first a control message (MessageTotalSize and Source Address)
*       It uses this data to know what data to expect next
*
*       a Get_from_EMAC is used to read the data payload from the fsls
*
*       Returns the CommVarsIndex for the user to be able to use it
*       it will enable him to know about the sender and about the state of
*       data sender ...
*/
Xuint32 Get_Frame_from_EMAC(Xuint8 *BuffPtr)
{
        Xuint32 FragmentSize;
        Xuint32 counter;
        Xuint32 CommVarsIndex;                          // this will hold the index into the array of structures CommVars

        Get_from_EMAC(Control_Buffer, 12);              // This will set the source address and TotalMessageSize
        Parse_Control_Message(Control_Buffer);  // Variables
                                                                                // Then we can use these variables to know what to expect from the
sender
                                                                                //
                                                                                // These variables are "Link" dependent
                                                                                // Implement a table in order to keep a variable for each communication
link

        CommVarsIndex = GetIndex(DestinationAddress);
                                                                                // now we know the index into the structure table

        CommVars[CommVarsIndex].MsgLength = TotalMessageSize;
        CommVars[CommVarsIndex].TypeofLastFrame = TypeofMessage;
```

```
	/*	We got a new message even though the old message did not arrive in totality, so we notify
		the user to flush the buffers he used to hold the frames of the non completed message !!  */
	if(CommVars[CommVarsIndex].RemainingSize != 0 && (CommVars[CommVarsIndex].TypeofLastFrame == 0 ||
CommVars[CommVarsIndex].TypeofLastFrame == 2))
		UserNotification = 1; // User should flush the buffers containing the fragments of the message
	else
		UserNotification = 0; // no problem

	/* Type 0 (Non Fragmented) Message */
	if(CommVars[CommVarsIndex].TypeofLastFrame == 0)
	{
		CommVars[CommVarsIndex].FragLength = TotalMessageSize;
					// also equal to CommVars[CommVarsIndex].MsgLength
		Get_from_EMAC(BuffPtr, CommVars[CommVarsIndex].FragLength);
		CommVars[CommVarsIndex].RemainingSize = 0;
		CommVars[CommVarsIndex].NbFragRecv = 1;
	}

	/* Type 2 (First Fragment) Message */
	else if(CommVars[CommVarsIndex].TypeofLastFrame == 2)
	{
		CommVars[CommVarsIndex].FragLength = MAX_L2_PAYLOAD_SIZE_BYTES;
		Get_from_EMAC(BuffPtr, CommVars[CommVarsIndex].FragLength);
		CommVars[CommVarsIndex].RemainingSize = CommVars[CommVarsIndex].MsgLength - MAX_L2_PAYLOAD_SIZE_BYTES;

		CommVars[CommVarsIndex].NbFragRecv = 1;
	}

	/* Type 3 (Internal Fragment) Message */
	else if(CommVars[CommVarsIndex].TypeofLastFrame == 3)
```

```
	{
		CommVars[CommVarsIndex].FragLength = MAX_L2_PAYLOAD_SIZE_BYTES;
		Get_from_EMAC(BuffPtr, CommVars[CommVarsIndex].FragLength);
		CommVars[CommVarsIndex].RemainingSize -= MAX_L2_PAYLOAD_SIZE_BYTES;
		CommVars[CommVarsIndex].NbFragRecv ++;
	}

	/* Type 4 (Last Fragment) Message */
	else if(CommVars[CommVarsIndex].TypeofLastFrame == 4)
	{
		CommVars[CommVarsIndex].FragLength = CommVars[CommVarsIndex].RemainingSize;
		Get_from_EMAC(BuffPtr, CommVars[CommVarsIndex].FragLength);
		CommVars[CommVarsIndex].RemainingSize = 0;
		CommVars[CommVarsIndex].NbFragRecv ++;
	}
	return CommVarsIndex; // for the user to check the variables
}
```

mb1_Functions.h

```
/*****************************************************************************
*
*       Simple Reliable Communication System over Ethernet _ Microblaze 0
*
*       Contains all the functions used by microblaze_1
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher-John Comis (University of Toronto)
*       at the following email address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/

#ifndef MB1_FUNCTIONS_H
#define MB1_FUNCTIONS_H

#include "mb1_GlobalVariables.h"

/****************************** Various Functions ******************************/

/*
```

```
* checks data when data is just a counter % 100 in the buffer
* this is used only for debugging purpose in case the buffer at the sender
* is filled with Fill_Buffer
*/
Xuint32 Check_Data(Xuint8 * Ptr, Xuint32 Size);

/*
* This will fill a buffer with known content for debugging by data consistency verification
*/
void Fill_Buffer(Xuint32 Size, Xuint8 *BuffPtr);

/*
* Takes the total message payload size
* returns the size of the last fragment payload
*/
Xuint32 Get_Last_Fragment_Size(Xuint32 TotalSz);

/*
* Takes the total message payload size
* Returns the number of fragments for this message
*/
Xuint32 Get_Number_of_Fragments(Xuint32 TotalSize);



/*
*       Will print the first Byte_Size bytes contained in Receive_from_EMAC_Buffer
*/
void Print_Receive_Content(Xuint32 Byte_Size);
```

```
/*
*       Takes an integer and returns a character
*       This is used to be able to print the MAC Address
*       in standard format_ex: 06-05-04-03-02-01
*/
char map(int x);

/*
*       Takes an integer and returns a character
*       This is used to be able to print the MAC Address
*       in standard format_ex: 06-05-04-03-02-01
*/
int getfirstone(int x);

/*
*       Takes a pointer to a MAC Address
*       and fills 'array' with the MAC Address in printable MAC standard format
*       ex: 06-05-04-03-02-01
*/
void fill_array(Xuint8 *Ptr);


/****************************************************************************/

#endif
```

mb1_Functions.c

```
/*****************************************************************************
*
*       Simple Reliable Communication System over Ethernet _ Microblaze 0
*
*       Contains all the functions source code used by microblaze_1
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher-John Comis (University of Toronto)
*       at the following email address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/

#include "mb1_Functions.h"
#include "mb1_GlobalVariables.h"

/*
*checks data when data is just a counter % 100 in the buffer
*/
Xuint32 Check_Data(Xuint8 * Ptr, Xuint32 Size)
{
```

```
	Xuint32 count;

	for(count = 0 ; count < Size ; count ++)
		if(*Ptr++ != count % 100)
			return 1; // Failure

	return 0; // Success
}

/*
* This will fill a buffer with known content for debugging by data consistency verification
*/
void Fill_Buffer(Xuint32 Size, Xuint8 *BuffPtr)
{
	Xuint32 count;
	Xuint8 * Ptr = BuffPtr;

	for(count = 0 ; count < Size ; count++)
	{
		*Ptr++ = count % 100;
	}
}


/*
* Takes the total message payload size
* returns the size of the last fragment payload
*/
Xuint32 Get_Last_Fragment_Size(Xuint32 TotalSz)
{
```

```
	if(TotalSz % MAX_L2_PAYLOAD_SIZE_BYTES == 0)
		return MAX_L2_PAYLOAD_SIZE_BYTES;
	else
		return TotalSz % MAX_L2_PAYLOAD_SIZE_BYTES;
}

/*
* Takes the total message payload size
* Returns the number of fragments for this message
*/
Xuint32 Get_Number_of_Fragments(Xuint32 TotalSize)
{
	if(TotalSize % MAX_L2_PAYLOAD_SIZE_BYTES == 0)
		return TotalSize / MAX_L2_PAYLOAD_SIZE_BYTES;
	else
		return 1 + TotalSize / MAX_L2_PAYLOAD_SIZE_BYTES;
}


/*
*	Will print the first Byte_Size bytes contained in Receive_from_EMAC_Buffer
*/
void Print_Receive_Content(Xuint32 Byte_Size)
{
	Xuint32 count;
	Xuint8 *Ptr = Receive_from_EMAC_Buffer;

	xil_printf("EMAC Microblaze Buffer Content %d\n\r", Byte_Size);
	xil_printf("----------------------------------------\n\r");
	for(count = 0 ; count < Byte_Size ; count ++)
```

```
	{
		xil_printf("%x", *Ptr++);
	}
}


/*
*	Takes an integer and returns a character
*	This is used to be able to print the MAC Address
*	in standard format_ex: 06-05-04-03-02-01
*/
char map(int x)
{
	switch(x)
	{
	case 0:
		return '0';
	case 1:
		return '1';
	case 2:
		return '2';
	case 3:
		return '3';
	case 4:
		return '4';
	case 5:
		return '5';
	case 6:
		return '6';
	case 7:
```

```
                return '7';
        case 8:
                return '8';
        case 9:
                return '9';
        case 10:
                return 'A';
        case 11:
                return 'B';
        case 12:
                return 'C';
        case 13:
                return 'D';
        case 14:
                return 'E';
        case 15:
                return 'F';
        }
}

/*
*       Takes an integer and returns a character
*       This is used to be able to print the MAC Address
*       in standard format_ex: 06-05-04-03-02-01
*/
int getfirstone(int x)
{
        switch(x)
        {
        case 1:
```

```
                return 0;
        case 2:
                return 1490;
        case 3:
                return 2980;
        case 4:
                return 4470;
        case 5:
                return 5960;
        case 6:
                return 7450;
        case 7:
                return 8940;
        }
}



/*
*       Takes a pointer to a MAC Address
*       and fills 'array' with the MAC Address in printable MAC standard format
*       ex: 06-05-04-03-02-01
*/
void fill_array(Xuint8 *Ptr)
{
        Xuint8 mycounter;
        for(mycounter = 0 ; mycounter < STANDARD_MAC_FORMAT_SIZE ; mycounter += 3)
        {
                array[mycounter+1] = map(*(Ptr) & 0xF);
```

```
                array[mycounter] = map( ( *(Ptr) & 0xF0 ) >> 4);
                Ptr++;
                if(mycounter<15) array[mycounter+2] = '-';
        }
}
```

mb1_GlobalVariables.h

```
/*****************************************************************************
*
*       This Header contains some definitions and global variables used
*       in the microblaze_1 system
*
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher John Comis of the University of Toronto
*       at the following address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/

#ifndef MB1_GLOBALVARIABLES_H
```

```
#define MB1_GLOBALVARIABLES_H

#include "MACAddresses.h"


/*****************************************************************************/


XIntc InterruptController;                  // InterruptController Instance (input is timer_intr_1)

#define MAX_MESSAGE_SIZE_BYTES              10000
#define CONTROL_BUFFER_SIZE_BYTES           12
#define MAC_ADDRESS_SIZE_BYTES              6
#defineMAX_L2_PAYLOAD_SIZE_BYTES            1490
#define STANDARD_MAC_FORMAT_SIZE            17

/********************************** Buffers ***********************************/

Xuint8 Receive_from_EMAC_Buffer[MAX_L2_PAYLOAD_SIZE_BYTES];     // Used to hold temporarily the payload of incoming frames
                                                                // user should read data from this buffer and place it in his
                                                                // application buffer accordingly
                                                                // demultiplexing can be done by looking at the destinatioAddress
                                                                // array which contains the MAC Address of the source that sent
                                                                // the message

Xuint8 Message_To_Send_Buffer[MAX_MESSAGE_SIZE_BYTES];  // temporary Buffer to hold the message payload the user wants to send
                                                                // User can redefine the size of this buffer to suit his application

Xuint8 Control_Buffer[CONTROL_BUFFER_SIZE_BYTES];                       // will hold temporarily the Control Message to then parse it
```

```
Xuint8 DestinationAddress[MAC_ADDRESS_SIZE_BYTES];                    // will hold the MAC Address of the source that sent us the message
                                                                      // Variable Set by : Parse_Control_Message Function

/********************************** Global Variables **********************************/

Xuint32 TotalMessageSize;                    // This will be in a table later

Xuint16 TypeofMessage;                              // this is to solve the problem of message loss

Xuint32 BusArbitration;

Xuint8 UserNotification;

char array[STANDARD_MAC_FORMAT_SIZE];  // just used as a buffer to store the MAC Address in standard
                                                                      // printable format ex : 06-05-04-03-02-01
                                                                      // do a xil_printf("%s", array); to print it
                                                                      // fill_array takes a pointer to the MAC Address and fills this array


/*****************************************************************************************/

typedef struct
{
Xuint32 TypeofLastFrame;

Xuint32 NbFragRecv;

Xuint32 MsgLength;

Xuint32 FragLength;
```

```
Xuint32 RemainingSize;

}CommunicationsVariables;

CommunicationsVariables CommVars[NUMBER_OF_BOARDS];

#endif
```

mb1_HandlerFunctions.h

```
/*****************************************************************************
*
*       Simple Reliable Communication System over Ethernet _ Microblaze 0
*
*       Contains all the functions used by microblaze_1
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher-John Comis (University of Toronto)
*       at the following email address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/
```

```
#ifndef MB1_HANDLERFUNCTIONS_H
#define MB1_HANDLERFUNCTIONS_H

#include "mb1_GlobalVariables.h"
#include "mb1_Functions.h"
#include "mb1_FSLTransferFunctions.h"
/*
*       This Handler is called when microblaze_0 interrupts microblaze_1
*       to tell it that data was received from another board and has to be
*       transfered
*
*       A small synchronization is done
*       and then we call Get_Frame_from_EMAC that will read a control message
*       and then read the payload
*
*       The data is contained in a buffer, it is the responsibility of the user to
*       copy the data into the application buffer using the Source Address and other
*       available parameters for demultiplexing
*
*       Note: CallBackRef is not used
*/
void Handler(void * CallBackRef);


#endif
```

mb1_HandlerFunctions.c

```
/*****************************************************************************
*
*       Reliable Communication System over Ethernet _ Microblaze 1
*
*       This Code contains the source code for the initialization of the microblaze_1
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher John Comis (University of Toronto)
*       at the following address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/

#include "mb1_HandlerFunctions.h"


/*
*       This Handler is called when microblaze_0 interrupts microblaze_1
*       to tell it that data was received from another board and has to be
*       transfered
*
*       A small synchronization is done
```

```
*       and then we call Get_Frame_from_EMAC that will read a control message
*       and then read the payload
*
*       The data is contained in a buffer, it is the responsibility of the user to
*       copy the data into the application buffer using the Source Address and other
*       available parameters for demultiplexing
*
*       Note: CallBackRef is not used
*/
void Handler(void * CallBackRef)
{

        Xuint32 temporary;
        Xuint32 counter;
        Xuint32 CommVarsIndex;
        Xuint8 * Ptr;
        Xuint32 errors = 0;
        Xuint32 FirstOne;
        Xuint8 flag;

        XIntc_Disable(&InterruptController, 0);

        Ptr = (Xuint8 *)Receive_from_EMAC_Buffer;

        /* Synchronization with the other microblaze */


        if(BusArbitration == 1)         // don't receive a frame and notify the microblaze_0
                                                // that it does not have the right to use the fsls
        {
```

```
microblaze_bwrite_datafsl(1, 1);
microblaze_bread_datafsl(temporary, 0);
}

else if(BusArbitration == 0)   // we can receive a frame
                                                    // notify the microblaze_0 to send us the frame
{

        microblaze_bwrite_datafsl(0, 1);
        microblaze_bread_datafsl(temporary, 0);

        /* The Handler Content Goes Here */

        // Getting the frame from the EMAC MicroBlaze and saving the content in Receive_from_Emac_Buffer
        // emac interrupts our microblaze before starting to send to us the frame we are waiting for

        // getting the frame from EMAC
        CommVarsIndex = Get_Frame_from_EMAC((Xuint8 *) Receive_from_EMAC_Buffer);


        /*****************************************************/
        /* Frame Reception: this is where your code starts */
        /*****************************************************/

/*      CommVarsIndex contains the index into the table of communication variables
        This table allows us to keep many variables since we are communicating with many
        boards at the same time and we need a set of variables for communication with
        each board.
        A similar table exists in microblaze_0 but it deals with sequence numbers and other
        variables for reliable transfer */
```

```
        xil_printf("Index                : %d\n\r", CommVarsIndex);

/*      CommVars[CommVarsIndex].MsgLength contains the total message length
        (not the payload length of the fragment frame !!) */
        xil_printf("MsgLength            : %d\n\r",CommVars[CommVarsIndex].MsgLength);

/*  CommVars[CommVarsIndex].FragLength contains the payload length of the received
        frame. We use it to know how much to read from Receive_from_EMAC_Buffer into
        the user defined buffer*/
        xil_printf("FragLength           : %d\n\r",CommVars[CommVarsIndex].FragLength);

/*  CommVars[CommVarsIndex].RemainingSize contains the remaining size (after this fragment)
        as part of the same message */
        xil_printf("RemSize              : %d\n\r",CommVars[CommVarsIndex].RemainingSize);

/*  CommVars[CommVarsIndex].NbFragRecv contains the number of fragments already received
        (as part of the same message) and including this fragment*/
        xil_printf("NbFragRecv           : %d\n\r",CommVars[CommVarsIndex].NbFragRecv);

/*  CommVars[CommVarsIndex].TypeofLastFrame contains the type of this frame
        Please Refer to the ProtocolDescription document for further information */
        xil_printf("Type                 : %d\n\r",CommVars[CommVarsIndex].TypeofLastFrame);


            /*      In case we were reading a fragmented message and we already have some frames buffered
                    and the on frame failed to reach us, the whole message fails. The user is notified of this
                    and has to flush the user defined buffer containing the received frames since they are no longer
                    usefull */
            if(UserNotification == 1)
                    xil_printf("Message Failed __ Flush Fragments Already Buffered !!\n\r");
```

```
/*      On the sender's side we are filling the 10,000 byte buffer with a counter % 100
        the getfirstone function will return to us the first expected byte value of the fragment
        according to the fragment number

        for example: fragment 1 will return 0 since the counter of Fill_Buffer starts from 0
        fragment 4 will return : 4470  since there are three 1490 fragments before it */

        FirstOne = getfirstone(CommVars[CommVarsIndex].NbFragRecv);

/*      Now we check the fragment's payload for consistency using the previous function */

        Ptr = (Xuint8 *)Receive_from_EMAC_Buffer;
        flag =0;

        for(counter = 0 ; counter < CommVars[CommVarsIndex].FragLength ; counter ++)
        {
                if(*Ptr ++ != ( FirstOne + counter )% 100  )
                {
                        xil_printf("errors %d\n\r", errors++);
                        flag =1;
                }
        }
        if(flag ==0)
        {       // content of the frame is valid
                xil_printf("OK : %d  %d\n\r", CommVars[CommVarsIndex].MsgLength, CommVars[CommVarsIndex].NbFragRecv);
        }
```

```
        /*      To copy the payload content into an appropriate user defined buffer
                Check the upper variables and just copy the first CommVars[CommVarsIndex].FragLength
                from Receive_from_EMAC_Buffer to your buffer */

        /*      Displaying the Sender's Address -- can be used for demultiplexing*/
                xil_printf("We received a frame from : Address%d\n\r", CommVarsIndex);




                /*************************************************/
                /* Frame Reception: this is where your code ends */
                /*************************************************/


                        /* Assuming data is consistent
                   tell the second microblaze that we are done
                   the value 50 means that everything is fine
                   otherwise we could use the value 40                                  */

                        microblaze_bwrite_datafsl(50, 1);

                // the 50 is just to make sure that
                // the receiver knows that we read correctly
        }

//      XIntc_Acknowledge(&InterruptController, 0);

        XIntc_Enable(&InterruptController, 0);
}
```

mb1_InitializationFunctions.h

```
/*****************************************************************************
*
*       Simple Reliable Communication System over Ethernet _ Microblaze 0
*
*       Contains all the functions used by microblaze_1
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher-John Comis (University of Toronto)
*       at the following email address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/

#ifndef MB1_INITIALIZATIONFUNCTIONS_H
#define MB1_INITIALIZATIONFUNCTIONS_H

#include "mb1_GlobalVariables.h"
#include "mb1_Functions.h"
#include "mb1_HandlerFunctions.h"
/*
*       Connects the Handler the the interrupt controller
```

```
*          and enables the interrupt controller
*/
XStatus Initialize_MicroBlaze_Interrupt_Controller();

/*
*
* This function is called from Initialize Parameters (which is the only one called from main)
* and hence there is no need to
* call it from main.
*
* Iinitializes the Communication Parameters Structure contents
* It assumes we have NUMBER_OF_BOARDS boards (check GlobalApp.h)
*/
void Initialize_CommVars();

/*
* Just Calls the upper two functions. Should be called from main
*/
XStatus Initialize_Parameters();

#endif
```

mb1_InitializationFunctions.c

```
/*****************************************************************************
*
*       Reliable Communication System over Ethernet _ Microblaze 1
*
*       This Code contains the source code for the initialization of the microblaze_1
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher John Comis (University of Toronto)
*       at the following address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/

#include "mb1_InitializationFunctions.h"

/*
*       Connects the Handler the the interrupt controller
*       and enables the interrupt controller
*/
XStatus Initialize_MicroBlaze_Interrupt_Controller()
{
```

```
	XStatus Result;

microblaze_enable_interrupts();

	Result = XIntc_Initialize(&InterruptController, 0);
	if (Result != XST_SUCCESS)
{
   xil_printf("Error in Xintc Initialization !!\n\r");
		return Result;
}
	else
   xil_printf("Xintc Initialization Succeeded !!\n\r");



	Result = XIntc_Connect(&InterruptController, 0, (XInterruptHandler)Handler, NULL);
if (Result != XST_SUCCESS)
{
   xil_printf("Error in Xintc Connect to Input !!\n\r");
		return Result;
}
	else
   xil_printf("Xintc Connect to Input Succeeded !!\n\r");



	Result = XIntc_SetOptions(&InterruptController, XIN_SVC_ALL_ISRS_OPTION);
	if (Result != XST_SUCCESS)
```

```
   {
                return Result;
      xil_printf("Error in Xintc SetOptions !!\n\r");
   }
        else
      xil_printf("Xintc SetOptions Succeeded !!\n\r");



        Result = XIntc_Start(&InterruptController, XIN_REAL_MODE);
        if (Result != XST_SUCCESS)
   {
                return Result;
      xil_printf("Error in Xintc Start !!\n\r");
   }
        else
      xil_printf("Xintc Start Succeeded !!\n\r");

        XIntc_Acknowledge(&InterruptController, 0);
        XIntc_Enable(&InterruptController, 0);

        return XST_SUCCESS;
}




/*
*
* This function is called from Initialize Parameters (which is the only one called from main)
```

```
* and hence there is no need to
* call it from main.
*
* Iinitializes the Communication Parameters Structure contents
* It assumes we have NUMBER_OF_BOARDS boards (check GlobalApp.h)
*/
void Initialize_CommVars()
{
	Xuint8 cntr;

	for(cntr=0 ; cntr < NUMBER_OF_BOARDS ; cntr++)
	{
		CommVars[cntr].TypeofLastFrame	= 0;
		CommVars[cntr].NbFragRecv		= 0;
		CommVars[cntr].MsgLength		= 0;
		CommVars[cntr].FragLength		= 0;
		CommVars[cntr].RemainingSize	= 0;
	}
}

/*
* Just Calls the upper two functions. Should be called from main
*/
XStatus Initialize_Parameters()
{
	XStatus Ret;

	/* priority to "reception" of frame */
	BusArbitration = 0;
```

```
	/* Initialize the communication variables with all other  boards on the system */
	Initialize_CommVars();
		xil_printf("Communications Variables Initialized !!\n\r");

	/* Initializing the interrupt controller so that microblaze_0 interrupts us */
	Ret = Initialize_MicroBlaze_Interrupt_Controller();
	if(Ret != XST_SUCCESS)
		xil_printf("Interrupt Controller Initialization Failure !!\n\r");
	else
		xil_printf("Interrupt Controller Initialization Success !!\n\r");

	return Ret;
}
```

mb1_SendFunctions.h

```
/*****************************************************************************
*
*	Simple Reliable Communication System over Ethernet _ Microblaze 0
*
*	Contains all the functions used by microblaze_1
*
*	Author:		Patrick E. Akl
*			American University of Beirut
*			August 2004
*
*	For any comments, suggestions or bug discovery, please contact me at the following
```

```
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher-John Comis (University of Toronto)
*       at the following email address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/

#ifndef MB1_SENDFUNCTIONS_H
#define MB1_SENDFUNCTIONS_H

#include "mb1_GlobalVariables.h"
#include "mb1_Functions.h"
/*
*       Takes Pointer to the Message and the length of the message in bytes
*
*       This function will send the whole message to microblaze_0 which will read it
*       and send it via Ethernet
*
*       After each fragment is sent, the function waits for a byte from microblaze_0
*       to know if the sending was succesfull
*
*       returns 0 for success or 1 for failure
*
*       NOTE: because fsls send in units of 4 byte words, we have to sometimes fill the last
*       two bytes with 0s
*       This procedure is synchronized with the algorithm of the receive function of
*       microblaze_0
*/
Xuint32 Send_Message_to_EMAC(Xuint8 *BuffPtr, Xuint32 Length_Bytes);
```

```
/*
*       Takes the message total size and a pointer to a MAC Address
*       it will send them to microblaze_0 in 3 4-byte fsl transfer with the following structure:
*
*       MessageTotalSize (4 bytes)___EMAC Address (6 bytes)
*
*       Note:   because fsl transfers are in units of 4 byte words,
*                       the last two bytes of the 3rd transfer are just filled with zeros
*
*/
void Send_Control_Message_to_EMAC(Xuint32 MessageTotalSize, Xuint8 * AddressPointer);

/*
*       Takes the total message size, a pointer to the start of the message and the destination address
*
*       It will first send a control message to microblaze_0, then it will send the data in fragments
*/
static Xuint8 Send_Control_and_Message_to_EMAC(Xuint32 MessagePayloadSz, Xuint8 *InputBuffer, Xuint8 *DestAddress);


/*
*       Takes the message total size, a pointer to the beginning of the message
*       and a pointer to the destination address.
*
*       We start sending a wake up byte to enable microblaze_0 to read what we'll send
*       and send the message in fragments via Ethernet
*
*       and then we send the control message telling microblaze_0 what to expect and the payload
*       of the message to send
```

```
*
*       NOTE :      This is the function to use in microblaze_1 to send a message
*                   The User should only use this function
*/
Xuint32 Send_EMAC(Xuint32 MessagePayloadSz, Xuint8 *InputBuffer, Xuint8 *DestAddress);
#endif
```

mb1_SendFunctions.c

```
/*****************************************************************************
*
*       Reliable Communication System over Ethernet _ Microblaze 1
*
*       This Code contains the source code for the Send Functions
*
*       Note : the user should use the send_Emac function
*
*       Author:     Patrick E. Akl
*                   American University of Beirut
*                   August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher John Comis (University of Toronto)
*       at the following address:
*                                   comis@eecg.toronto.edu
*
*****************************************************************************/
```

```
#include "mb1_SendFunctions.h"

/*
*	Takes Pointer to the Message and the length of the message in bytes
*
*	This function will send the whole message to microblaze_0 which will read it
*	and send it via Ethernet
*
*	After each fragment is sent, the function waits for a byte from microblaze_0
*	to know if the sending was succesfull
*
*	returns 0 for success or 1 for failure
*
*	NOTE: because fsls send in units of 4 byte words, we have to sometimes fill the last
*	two bytes with 0s
*	This procedure is synchronized with the algorithm of the receive function of
*	microblaze_0
*/
Xuint32 Send_Message_to_EMAC(Xuint8 *BuffPtr, Xuint32 Length_Bytes)
{
	Xuint32 NumberofFragments;
	Xuint32 counter1;
	Xuint32 counter2;
	Xuint32 SizeofLastFragment;
	Xuint32 Temp;
	Xuint8 * Pointer;
	Xuint8 * Ptr = BuffPtr;
	Xuint32 Val;
```

```
        NumberofFragments = Get_Number_of_Fragments(Length_Bytes);
        SizeofLastFragment = Get_Last_Fragment_Size(Length_Bytes);

// Deal With the internal Fragments
        for(counter1 = 0 ; counter1 < (NumberofFragments - 1); counter1 ++)
        {                               // rounded to lower integer = 372 = 1490 / 4
                for(counter2 = 0 ; counter2 < MAX_L2_PAYLOAD_SIZE_BYTES / 4 ; counter2 ++)
                {
                        Pointer = (Xuint8 *)&Temp;

                        *Pointer++ = *Ptr++;
                        *Pointer++ = *Ptr++;
                        *Pointer++ = *Ptr++;
                        *Pointer++ = *Ptr++;

                        microblaze_bwrite_datafsl(Temp, 1);
                }       // wrote the first 1488 bytes

                Pointer = (Xuint8 *)&Temp;

                *Pointer++ = *Ptr++;
                *Pointer++ = *Ptr++;

                *Pointer++ = 0;
                *Pointer++ = 0;

                microblaze_bwrite_datafsl(Temp, 1); // sent last two bytes of fragment
                /* Here We sent a Fragment So we wait for an ACK from the EMAC MicroBlaze */
                microblaze_bread_datafsl(Val, 0);
```

```
                if(Val == 1) // 1 for failure __ 0 for success
                        return 1; // failure
        }

// Deal With the Last Fragment
        // Deal with the first full words
        for(counter1 = 0 ; counter1 < (SizeofLastFragment / 4) ; counter1 ++)
        {
                Pointer = (Xuint8 *)&Temp;

                *Pointer++ = *Ptr++;
                *Pointer++ = *Ptr++;
                *Pointer++ = *Ptr++;
                *Pointer++ = *Ptr++;

                microblaze_bwrite_datafsl(Temp, 1);
        }
        // Now Deal With the Remaining Bytes
        if(SizeofLastFragment % 4 == 1)
        {       // Still have 1 byte to send
                Pointer = (Xuint8 *)&Temp;
                *Pointer++ = *Ptr++;
                *Pointer++ = 0;
                *Pointer++ = 0;
                *Pointer++ = 0;
                microblaze_bwrite_datafsl(Temp, 1);
        }

        if(SizeofLastFragment % 4 == 2)
        {       // Still have 2 bytes to send
```

```
		Pointer = (Xuint8 *)&Temp;
		*Pointer++ = *Ptr++;
		*Pointer++ = *Ptr++;
		*Pointer++ = 0;
		*Pointer++ = 0;
		microblaze_bwrite_datafsl(Temp, 1);
	}

	if(SizeofLastFragment % 4 == 3)
	{	// Still have 3 bytes to send
		Pointer = (Xuint8 *)&Temp;

		*Pointer++ = *Ptr++;
		*Pointer++ = *Ptr++;
		*Pointer++ = *Ptr++;
		*Pointer++ = 0;
		microblaze_bwrite_datafsl(Temp, 1);
	}

		/* Here We sent a Fragment So we wait for an ACK from the EMAC MicroBlaze */
		microblaze_bread_datafsl(Val, 0);
		if(Val == 1)
			return 1;  // failure
	return 0 ; // success
}

/*
*	Takes the message total size and a pointer to a MAC Address
*	it will send them to microblaze_0 in 3 4-byte fsl transfer with the following structure:
*
```

```
*       MessageTotalSize (4 bytes)___EMAC Address (6 bytes)
*
*       Note:   because fsl transfers are in units of 4 byte words,
*                       the last two bytes of the 3rd transfer are just filled with zeros
*
*/
void Send_Control_Message_to_EMAC(Xuint32 MessageTotalSize, Xuint8 * AddressPointer)
{
        Xuint32 Temp;
        Xuint8 * Ptr = Control_Buffer;
        Xuint8 * AddrPtr = AddressPointer;

        Xuint8 * Pointer;

        *(Xuint32 *)Ptr = MessageTotalSize;          // Writing Message Total Size

        Ptr += 4;

   *Ptr++ = *AddrPtr++;                              // Writing destination address
   *Ptr++ = *AddrPtr++;
   *Ptr++ = *AddrPtr++;
   *Ptr++ = *AddrPtr++;
   *Ptr++ = *AddrPtr++;
   *Ptr++ = *AddrPtr++;

        Ptr = Control_Buffer;

        microblaze_bwrite_datafsl(*(Xuint32 *)Ptr,1);
        Ptr += 4;
        microblaze_bwrite_datafsl(*(Xuint32 *)Ptr,1);
```

```
	Ptr += 4;

	Pointer = (Xuint8 *)&Temp;

	*Pointer ++ = *Ptr ++;
	*Pointer ++ = *Ptr ++;
	*Pointer ++ = 0;
	*Pointer ++ = 0;

	microblaze_bwrite_datafsl(Temp,1);
}

/*
*	Takes the total message size, a pointer to the start of the message and the destination address
*
*	It will first send a control message to microblaze_0, then it will send the data in fragments
*/
static Xuint8 Send_Control_and_Message_to_EMAC(Xuint32 MessagePayloadSz, Xuint8 *InputBuffer, Xuint8 *DestAddress)
{
	if(MessagePayloadSz > MAX_MESSAGE_SIZE_BYTES)		// Valid Message Size
		return 1; // failure

		// This is to Initialize the Second MicroBlaze System
	Send_Control_Message_to_EMAC(MessagePayloadSz, DestAddress);

		// This is sending the Message Fragments to the second MicroBlaze
	return  Send_Message_to_EMAC(InputBuffer ,MessagePayloadSz);

		// check if we keep on sending
}
```

```
/*
*       Takes the message total size, a pointer to the beginning of the message
*       and a pointer to the destination address.
*
*       We start sending a wake up byte to enable microblaze_0 to read what we'll send
*       and send the message in fragments via Ethernet
*
*       and then we send the control message telling microblaze_0 what to expect and the payload
*       of the message to send
*/
Xuint32 Send_EMAC(Xuint32 MessagePayloadSz, Xuint8 *InputBuffer, Xuint8 *DestAddress)
{
        Xuint32 ReturnValue;

        BusArbitration = 1;     // Priority to Sending a Message

        /* Send a Wake - Up Byte to the second MicroBlaze                                                                */
        /* This will let the code exit the while loop of the non blocking read and receive the frame */
        microblaze_bwrite_datafsl(60, 1);

        /* It will then read the information we are sending now */
        /* and send it to the destination using OPB_EMAC            */

        ReturnValue = Send_Control_and_Message_to_EMAC(MessagePayloadSz, InputBuffer, DestAddress);

        BusArbitration = 0;

        return ReturnValue;
}
```

Sniffer Node Implementation for Network Monitoring

The hardware configuration of the network sniffer node is very similar to that of the system running on a single MicroBlaze. In fact, the difference between the two implementations is mainly in the software configuration, since the role of the sniffer node is configured in promiscuous mode and only reads frames on the network. The main hardware difference between the sniffer node and the single MicroBlaze system is that the sniffer node does not need a timer counter because it does not deal frame transmission and the timer counter was only used for retransmission purpose.

Hardware Configuration

The hardware configuration consists of the following modules:

Peripheral	HW Ver	Instance	Base Address	High Address	Min
microblaze	2.10.a	microblaze_0			
opb_mdm	2.00.a	debug_module	0x80010000	0x800100ff	0x100
lmb_bram_if_cntlr	1.00.b	dlmb_cntlr	0x00000000	0x0000ffff	0x800
lmb_bram_if_cntlr	1.00.b	ilmb_cntlr	0x00000000	0x0000ffff	0x800
bram_block	1.00.a	lmb_bram			
opb_uartlite	1.00.b	RS232	0x80010100	0x800101ff	0x100
opb_gpio	3.01.a	opb_gpio_0	0x80010200	0x800103ff	0x1FF
opb_intc	1.00.c	opb_intc_0	0x80010500	0x8001051f	0x20
opb_ethernet	1.00.m	opb_ethernet_0	0x80014000	0x80017fff	0x4000

- MicroBlaze_0: The processor.
- OPB: On Chip Peripheral Bus used to interconnect the processor to the various peripheral devices used.
- OPB GPIO: peripheral general purpose input output device connected to the MicroBlaze using the OPB, used to take input from a DIP Switch and drive output to a user LED.
- OPB MDM: peripheral Microprocessor debug module connected to the MicroBlaze using the OPB.
- OPB RS232: peripheral serial link device connected to the MicroBlaze using the OPB and used to drive the output serial link that is used to print statements for debugging and verification on a HyperTerminal on a computer.
- OPB ETHERNET: peripheral Ethernet controller device connected to the MicroBlaze using the OPB.

- OPB INTC: peripheral interrupt controller device connected to the MicroBlaze using the OPB, and with the processor interrupt signal directly connected to the processor interrupt signal of the MicroBlaze. It takes as an input the interrupt output of the Ethernet controller.

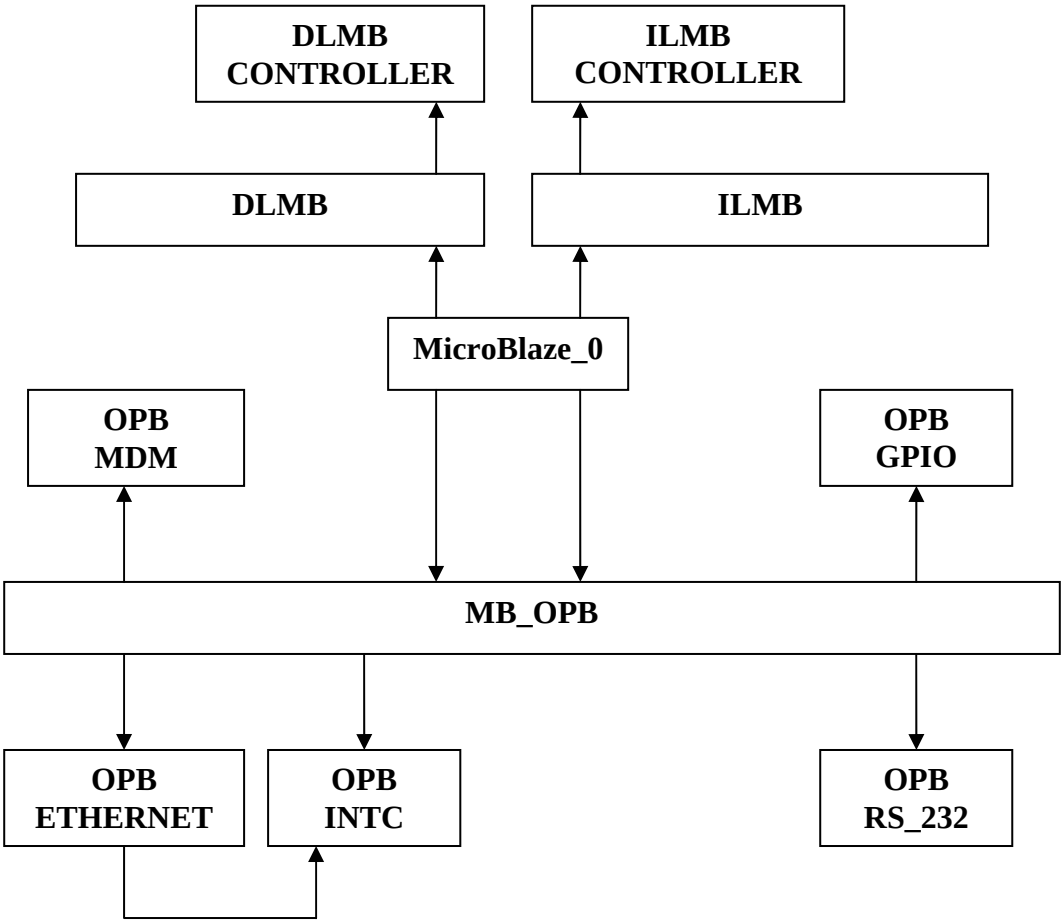
- DLMB: Data Local Memory Bus connected to the MicroBlaze.

- ILMB: Instruction Local Memory Bus connected to the MicroBlaze.

- DLMB CONTROLLER: Data Local Memory Bus Controller connected to DLMB.

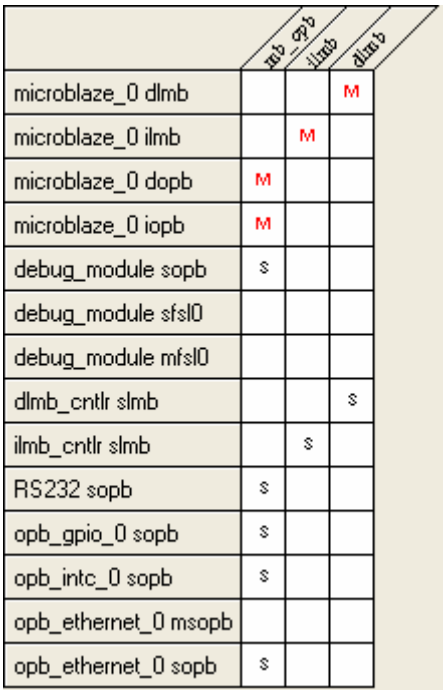
- ILMB CONTROLLER: Instruction Local Memory Bus Controller connected to ILMB.

Here is a schematic of the hardware modules in the sniffer system:



Bus Connections

Note that **M** means master and **s** means slave.



	mb_opb	ilmb	dlmb
microblaze_0 dlmb			M
microblaze_0 ilmb		M	
microblaze_0 dopb	M		
microblaze_0 iopb	M		
debug_module sopb	S		
debug_module sfsl0			
debug_module mfsl0			
dlmb_cntlr slmb			S
ilmb_cntlr slmb		S	
RS232 sopb	S		
opb_gpio_0 sopb	S		
opb_intc_0 sopb	S		
opb_ethernet_0 msopb			
opb_ethernet_0 sopb	S		

Port Connections

In this sniffer implementation, the interrupt output of the **opb_ethernet** module **emac_intr** is connected to the interrupt input of the interrupt controller **intc**. This is used in order to notify the MicroBlaze that a frame was received and is ready to be read.

Note that all the ports connected with External Scope are connected to external pins on the Multimedia Board. These connections are described in the .UCF file.

Instance	Port Name	Net Name	Polarity	Scope	Range	Class	Sensitivity
system	PHY_slew1	net_vcc	OUT	External			
system	PHY_slew2	net_vcc	OUT	External			
system	sys_rst	sys_rst_s	input	External			
system	sys_clk	sys_clk_s	input	External		CLK	
system	RS232_req_t...	net_gnd	OUTPUT	External			
opb_intc_0	Irq	IntConnection2	O	Internal		INTERRUPT	
opb_intc_0	Intr	emac_intr	I	Internal		INTERRUPT	
opb_gpio_0	GPIO_IO	opb_gpio_0_GPIO_IO	IO	External	[0:1]		
opb_ether...	IP2INTC_Irpt	emac_intr	O	Internal		INTERRUPT	LEVEL_HIGH
opb_ether...	PHY_tx_er	opb_ethernet_0_PHY_tx_er	O	External			
opb_ether...	PHY_tx_en	opb_ethernet_0_PHY_tx_en	O	External			
opb_ether...	PHY_Mii_clk	opb_ethernet_0_PHY_Mii_clk	IO	External			
opb_ether...	PHY_tx_data	opb_ethernet_0_PHY_tx_data	O	External	[3:0]		
opb_ether...	PHY_tx_clk	opb_ethernet_0_PHY_tx_clk	I	External			
opb_ether...	PHY_rx_er	opb_ethernet_0_PHY_rx_er	I	External			
opb_ether...	PHY_rx_data	opb_ethernet_0_PHY_rx_data	I	External	[3:0]		
opb_ether...	PHY_rx_clk	opb_ethernet_0_PHY_rx_clk	I	External			
opb_ether...	PHY_dv	opb_ethernet_0_PHY_dv	I	External			
opb_ether...	PHY_crs	opb_ethernet_0_PHY_crs	I	External			
opb_ether...	PHY_col	opb_ethernet_0_PHY_col	I	External			
opb_ether...	PHY_Mii_data	opb_ethernet_0_PHY_Mii_data	IO	External			
microblaze_0	CLK	sys_clk_s	I	Internal		CLK	
microblaze_0	INTERRUPT	IntConnection2	I	Internal		INTERRUPT	LEVEL_HIGH
mb_opb	OPB_Clk	sys_clk_s	I	Internal		CLK	
mb_opb	SYS_Rst	sys_rst_s	I	Internal			
ilmb	SYS_Rst	sys_rst_s	I	Internal			
ilmb	LMB_Clk	sys_clk_s	I	Internal		CLK	
dlmb	LMB_Clk	sys_clk_s	I	Internal		CLK	
dlmb	SYS_Rst	sys_rst_s	I	Internal			
debug_module	OPB_Clk	sys_clk_s	I	Internal		CLK	
RS232	OPB_Clk	sys_clk_s	I	Internal		CLK	
RS232	TX	RS232_TX	O	External			
RS232	RX	RS232_RX	I	External			

External Port Connections: .UCF File

Net sys_clk PERIOD = 37037 ps;
Net sys_clk LOC=AH15;
Net sys_rst LOC=AH7;

Net RS232_RX LOC=C8;
Net RS232_TX LOC=C9;
Net RS232_req_to_send LOC=B8;

Net opb_gpio_0_GPIO_IO<1> LOC=B27;
Net opb_gpio_0_GPIO_IO<0> LOC=D10;

Net PHY_slew1 LOC=G16;
Net PHY_slew2 LOC=C16;

Net opb_ethernet_0_PHY_crs LOC=F20;
Net opb_ethernet_0_PHY_col LOC=C23;
Net opb_ethernet_0_PHY_tx_data<3> LOC=C22;
Net opb_ethernet_0_PHY_tx_data<2> LOC=B20;

```
Net opb_ethernet_0_PHY_tx_data<1> LOC=B21;
Net opb_ethernet_0_PHY_tx_data<0> LOC=G20;
Net opb_ethernet_0_PHY_tx_en LOC=G19;
Net opb_ethernet_0_PHY_tx_clk LOC=H16;
Net opb_ethernet_0_PHY_tx_er LOC=D21;
Net opb_ethernet_0_PHY_rx_er LOC=D22;
Net opb_ethernet_0_PHY_rx_clk LOC=C17;
Net opb_ethernet_0_PHY_dv LOC=B17;
Net opb_ethernet_0_PHY_rx_data<0> LOC=B16;
Net opb_ethernet_0_PHY_rx_data<1> LOC=F17;
Net opb_ethernet_0_PHY_rx_data<2> LOC=F16;
Net opb_ethernet_0_PHY_rx_data<3> LOC=D16;
Net opb_ethernet_0_PHY_Mii_clk LOC=D17;
Net opb_ethernet_0_PHY_Mii_data LOC=A17;
```

User Tutorial: Setting up a Sniffer Board

Introduction*:*

The Sniffer reads all the frames on the network; this is done by selecting the EMAC Promiscuous Mode Option. It parses the frame and message headers and displays some information. It can act in two modes:

Light Monitoring Mode: the sniffer displays the destination and source MAC Addresses of the frame as well as the type of message (refer to the table at the end of the document) and the total message (Layer 2) Length.

Heavy Monitoring Mode: the sniffer displays all the information contained in the frame and message headers as well as the first and last byte of each frame's payload for data consistency verification.

The User can select any of these modes using the DIP Switch SW0.

Preparation*:*

Read the documentation on the sniffer board as well as on the protocol description to make sure you understand well the experiment.

Equipment Needed*:*

- A Multimedia Board that will be used as a sniffer.
- Ethernet cables and a hub.

Setup*:*

19. Connect the Board you selected to act as a sniffer to the Hub or other shared medium link. Note that the Sniffer Board would not operate on a switch since the switch will direct the frames only to the proper receiver.

20. Set up a terminal window on your computer, this will be used to display the monitoring messages of the Sniffer. Connect the serial port of your computer to the serial port of the Sniffer Board.

21. In the Sniffer directory, double click on the XPS Icon. Under the Applications Menu, open the source file. Compile the source program, update the bitsream and download the executable onto the board.

22. Check your HyperTerminal screen, it should indicate that the automatic setup for the sniffer was successful. Check the User Led on the board it should turn on.

23. Toggle the User DIP Switch SW0 a few times and watch what happens on the HyperTerminal: You are effectively changing the configuration of what data to output to the screen when the sniffer reads a frame on the network.

24. Now you are ready to connect the other boards to the network and use the sniffer to monitor the network transactions happening on the network.

Important Notes*:*

Because the Sniffer is displaying data via the slow serial port, overflow can happen because of very fast frame rates in the system. In this case, the output of the sniffer **would not** reflect the real content of the frames. Therefore, the sniffer should be only used to monitor systems with low frame rates. This is also because the user would not be able to read anyways what's being displayed in case of very high frame rate.

Modifying What to Display

In case your HyperTerminal screen was being overflowed by too many fragments as part of a same message and you wanted to remove some extra xil_printf statement of variables irrelevant to you, you can easily comment out those in the Print_Message_Fields () and Print_Frame_Fields () in the Sniffer_DebugFunctions.c file.

Future Work*:*

It would be nice and very easy to modify this sniffer to keep track of some statistics like the number of frames exchanged between nodes x and y, the average number of retransmissions… This can be done by having some tables indexed by the MAC Addresses of the sender board and / or destination board. Keeping a log of the statistics instead of displaying them would let the sniffer

Software Configuration and Source Code

SnifferMain.c

```
/*****************************************************************************
*
*       SnifferMain.c
*       Simple Network Sniffer
*
*       This code Setup a board to act in promiscuous mode to monitor the network
*       transactions. It doesn't affect the network since it is just listening to the
*       frames exchanged on common media and displaying some frame and message layer
*       variables read from the frame.
*
*       NOTE 1:
*       The user can select two modes of display using the DIP Switch. One mode is heavy and
*       displays all the fields in the frame and message header as well as the first and last
*       bytes of the data payload for data consistency verifications
*       The second mode will simply display the type of the message as well as the destination
*       and source MAC Addresses.
*
*       Types of Messages :
*
*       0 _ Non Fragmented Message
*       1 _ ACK Message
*       2 _     First Fragment Frame
*       3 _     Internal Fragment Frame
*       4 _     Last Fragment Frame
*
```

```
*       NOTE 2:
*       This Simple Network Sniffer would not work on a Switch since the switch would direct
*       the frames to their destination boards only. Hence this sniffer would only be used on a
*       HUB Based Network System (Or on any other shared media Network System)
*
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher-John Comis (University of Toronto)
*       at the following email address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/

/*****************************  Include Files *********************************/

#include "xemac.h"                      // Ethernet
#include "xparameters.h"
#include "xstatus.h"

#include "mb_interface.h"       // MicroBlaze
#include "xintc.h"                      // for interrupt control
#include "xgpio.h"                      // for the switch

#include "Sniffer_GlobalVariables.h"            // contains the global variables shared among the 5 files
```

```
#include "Sniffer_Handlers.h"                    // contains the handlers functions
#include "Sniffer_DebugFunctions.h"          // contains functions for debugging using UArt
#include "Sniffer_Initializations.h"    // contains the functions that initialize our instances

#include "Sniffer_Handlers.c"                    // contains the handlers functions
#include "Sniffer_DebugFunctions.c"          // contains functions for debugging using UArt
#include "Sniffer_Initializations.c"    // contains the functions that initialize our instances

/*********************************    MAIN FUNCTION      *******************************/

int main()
{
        xil_printf("\n\n\n\n\n\n\r");

        /* Initializations ** EMAC and INTERRUPTS and GPIO */
        Initialize_Parameters();

        xil_printf("\n\rNetwork Transactions Sniffer Node\n\r");
        xil_printf("_____________________________________\n\n\n\r");

        /* Loop to keep sniffing the network */
   while (1)
   {
                if(XGpio_DiscreteRead(&Gpio) >= 2)
                {
                        for(debouncing = 0 ; debouncing < 100 ; debouncing ++);

                        if(XGpio_DiscreteRead(&Gpio) >= 2)
                        {
                                if(DegreeofInformation == 0)
```

```
                                        {
                                                DegreeofInformation = 1;

                                                xil_printf("Heavy Monitoring Mode\n\n\r");
                                        }
                                }
                        }
                else if (XGpio_DiscreteRead(&Gpio) < 2)
                        {
                                for(debouncing = 0 ; debouncing < 100 ; debouncing ++);

                                if(XGpio_DiscreteRead(&Gpio) < 2)
                                {
                                        if(DegreeofInformation == 1)
                                        {
                                                DegreeofInformation = 0;

                                                xil_printf("Light Monitoring Mode\n\n\r");
                                        }
                                }
                        }
    }

        return 0;
}
```

Sniffer_Initializations.h

```
/*****************************************************************************
*
*       Sniffer_Initializations.h
*       Simple Network Sniffer __ Initialization Functions
*
*       These functions will initialize the GPio, the EMAC Controller, as well as the
*       Interrupt Controller
*
*       In Main only Initialize_Parameters() is called. it turns the led in case everything
*       Was properly initialized and returns XST_SUCCESS
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher-John Comis (University of Toronto)
*       at the following email address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/

#ifndef SNIFFER_INITIALIZATIONS_H
#define SNIFFER_INITIALIZATIONS_H
```

```
#include "Sniffer_GlobalVariables.h"
#include "Sniffer_Handlers.h"

/*
* Initializes the GPio
*       DIP Switch is used to select the amount of information to sniff
*       Led is an output
*/
static XStatus Initialize_GPio();


/*
* Setup Interrupt System
* EMAC has input 0 of the interrupt controller
*/

static XStatus SetupInterruptSystem(XEmac *EmacPtr);

/*
* Initialize the EMAC Module
* It is configured in promiscuous mode
* for network monitoring
*/
static XStatus InitializeEmac();

/*
* This function should be called from main
* It calls all the other functions.
*/
void Initialize_Parameters();
```

#endif

Sniffer_Initializations.c

```
/*****************************************************************************
*
*       Sniffer_Initializations.c
*       Simple Network Sniffer __ Initialization Functions
*
*       These functions will initialize the GPio, the EMAC Controller, as well as the
*       Interrupt Controller
*
*       In Main only Initialize_Parameters() is called. it turns the led in case everything
*       Was properly initialized and returns XST_SUCCESS
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher-John Comis (University of Toronto)
*       at the following email address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/
```

```
#include "Sniffer_GlobalVariables.h"
#include "Sniffer_Handlers.h"

/*
* Initializes the GPio
*       DIP Switch is used to select the amount of information to sniff
*       Led is an output
*/
static XStatus Initialize_GPio()
{
        XStatus Status;
   Status = XGpio_Initialize(&Gpio, XPAR_OPB_GPIO_0_DEVICE_ID);
   if (Status != XST_SUCCESS)
   {
      xil_printf("GPIO initialization error !!\n\r");
   }
        else
                xil_printf("GPIO initialization succeeded !!\n\r");
   /* Set the direction for all signals to be inputs except the LED
    * outputs */
   XGpio_SetDataDirection(&Gpio, ~LED);

        return Status;
}


/*
* Setup Interrupt System
* EMAC has input 0 of the interrupt controller
*/
```

```
static XStatus SetupInterruptSystem(XEmac *EmacPtr)                    /* Setup Interrupt System */
{
   XStatus Result;

   microblaze_enable_interrupts();

   Result = XIntc_Initialize(&InterruptController, INTC_DEVICE_ID);
         if (Result != XST_SUCCESS)
   {
      xil_printf("Error in Xintc Initialization !!\n\r");
                  return Result;
   }
         else
      xil_printf("Xintc Initialization Succeeded !!\n\r");


         Result = XIntc_Connect(&InterruptController, INTC_INPUT_EMAC, (XInterruptHandler)XEmac_IntrHandlerFifo, EmacPtr);
   if (Result != XST_SUCCESS)
   {
                  return Result;
      xil_printf("Error in Xintc Connect to EMAC!!\n\r");
   }
         else
      xil_printf("Xintc Connect Succeeded to EMAC!!\n\r");


         Result = XIntc_SetOptions(&InterruptController, XIN_SVC_ALL_ISRS_OPTION);
         if (Result != XST_SUCCESS)
   {
```

```
                return Result;
        xil_printf("Error in Xintc SetOptions !!\n\r");
    }
        else
        xil_printf("Xintc SetOptions Succeeded !!\n\r");


        Result = XIntc_Start(&InterruptController, XIN_REAL_MODE);
        if (Result != XST_SUCCESS)
    {
                return Result;
        xil_printf("Error in Xintc Start !!\n\r");
    }
        else
        xil_printf("Xintc Start Succeeded !!\n\r");


    XIntc_Enable(&InterruptController, INTC_INPUT_EMAC);

    return XST_SUCCESS;

}


/*
* Initialize the EMAC Module
* It is configured in promiscuous mode
* for network monitoring
*/
static XStatus InitializeEmac()
```

```
{
        XEmac_Config *ConfigPtr;
        XStatus Result;
        Xuint32 DeviceId;
        DeviceId = 0;
        EmacPtr = & Emac;

   ConfigPtr = XEmac_LookupConfig(DeviceId);
   ConfigPtr->IpIfDmaConfig = XEM_CFG_NO_DMA;

   Result = XEmac_Initialize(EmacPtr, DeviceId);
   if (Result != XST_SUCCESS)
   {
                xil_printf("Error in EMAC Initialization !!\n\r");
      return Result;
   }
        else
                xil_printf("Initialization Succeeded !!\n\r");


   Result = XEmac_SelfTest(EmacPtr);
   if (Result != XST_SUCCESS)
   {
                xil_printf("Self Test Error !!\n\r");
      return Result;
   }
        else
                xil_printf("Self Test Succeeded !!\n\r");

   Result = XEmac_SetOptions(EmacPtr, TEST_1_OPTIONS);
```

```
if (Result != XST_SUCCESS)
{
                xil_printf("Set Options Error !!\n\r");
   return Result;
}
      else
                xil_printf("Set Options Succeeded !!\n\r");

Result = XEmac_SetMacAddress(EmacPtr, LocalAddress);
if (Result != XST_SUCCESS)
{
                xil_printf("Set MAC Address Error !!\n\r");
   return Result;
}
      else
                xil_printf("Set MAC Address Succeeded !!\n\r");

XEmac_SetFifoSendHandler(EmacPtr, EmacPtr, FifoSendHandler);
XEmac_SetFifoRecvHandler(EmacPtr, EmacPtr, FifoRecvHandler);
XEmac_SetErrorHandler(EmacPtr, EmacPtr, ErrorHandler);


      XEmac_ClearStats(EmacPtr);
Result = XEmac_Start(EmacPtr);
if (Result != XST_SUCCESS)
{
                xil_printf("EMAC Start Error !!\n\r");
   return Result;
}
      else
```

```
                xil_printf("EMAC Start Success !!\n\r");

        return XST_SUCCESS;
}


/*
* This function should be called from main
* It calls all the other functions.
*/
void Initialize_Parameters()
{
        XStatus Result;

        xil_printf("Initializing Sniffer\n\r");
        xil_printf("_____________________\n\n\r");

        DegreeofInformation = 1; // more information

        Result = InitializeEmac();
        if (Result != XST_SUCCESS)
   {
                xil_printf("EMAC Initialization Error !!\n\r");
      return;
   }
        else
                xil_printf("EMAC Initialization Success !!\n\r");


   Result = SetupInterruptSystem(EmacPtr);
```

```
   if (Result != XST_SUCCESS)
   {
                xil_printf("Set Interrupt System Error !!\n\r");
      return;
   }
        else
                xil_printf("Set Interrupt System Success !!\n\r");


        Result = Initialize_GPio();
   if (Result != XST_SUCCESS)
   {
                xil_printf("GPio Initialization Error !!\n\r");
      return;
   }
        else
                xil_printf("GPio Initialization Success !!\n\r");
}
```

Sniffer_Handlers.h

```
/*****************************************************************************
*
*       Sniffer_Handlers.h
*       Simple Network Sniffer __ Handler Functions
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher-John Comis (University of Toronto)
*       at the following email address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/

#ifndef SNIFFER_HANDLERS_H
#define SNIFFER_HANDLERS_H

#include "Sniffer_GlobalVariables.h"
#include "Sniffer_DebugFunctions.h"

/*
*       This function will read all frames on the network and will
*       display the main parameters of the messages
```

```
*
*       We also check the state of the Gpio to check whether to print
*       some information or all the fields
*/
static void FifoRecvHandler(void *CallBackRef);

/*
* Checks for errors.
*/
static void FifoSendHandler(void *CallBackRef);

/*
* Called in case of Errors to reset the system
*/
static void ErrorHandler(void *CallBackRef, XStatus Code);

#endif
```

Sniffer_Handlers.c

```
/*****************************************************************************
*
*       Sniffer_Handlers.c
*       Simple Network Sniffer __ Handler Functions
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher-John Comis (University of Toronto)
*       at the following email address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/


#include "Sniffer_GlobalVariables.h"
#include "Sniffer_DebugFunctions.h"

/*
*       This function will read all frames on the network and will
*       display the main parameters of the messages
*
*       We also check the state of the Gpio to check whether to print
```

```
*       some information or all the fields
*/
static void FifoRecvHandler(void *CallBackRef)
{
        XStatus Result;

        if(XGpio_DiscreteRead(&Gpio) >= 2)                  // no need for debouncing
        {
                        DegreeofInformation = 1;
                }
        else if (XGpio_DiscreteRead(&Gpio) < 2)    // no need for debouncing
                {
                        DegreeofInformation = 0;
                }


        XIntc_Disable(&InterruptController, INTC_INPUT_EMAC);

        FrameLen = XEM_MAX_FRAME_SIZE;
        Result = XEmac_FifoRecv((XEmac *)CallBackRef, RecvBuffer, &FrameLen);

        if (Result != XST_SUCCESS)
        {
                xil_printf("Fifo Receive Error\n\r");

                XIntc_Enable(&InterruptController, INTC_INPUT_EMAC);
                return;
        }
        else    // Success In Reception
        {
```

```
                        xil_printf("\n\r");
                        Print_Frame_Fields(RecvBuffer, FrameLen, DegreeofInformation);
                        Print_Message_Fields(RecvBuffer, FrameLen - 18, DegreeofInformation);
        }

        XIntc_Enable(&InterruptController, INTC_INPUT_EMAC);
}


/*
* Checks for errors.
*/
static void FifoSendHandler(void *CallBackRef)
{
   XEmac_GetStats((XEmac *)CallBackRef, &Stats);

        if (Stats.XmitLateCollisionErrors || Stats.XmitExcessDeferral)
   {
                xil_printf("Late Collision Number : %d\n\r", Stats.XmitLateCollisionErrors);
                xil_printf("Late Deferrals Number : %d\n\r", Stats.XmitExcessDeferral);
                xil_printf("Error in FIFO Sending !!\n\r");
   }

        XEmac_ClearStats((XEmac *)CallBackRef);
}


/*
* Called in case of Errors to reset the system
*/
```

```
static void ErrorHandler(void *CallBackRef, XStatus Code)
{
        xil_printf("Starting FIFO Error Handler\n\r");
   if (Code == XST_RESET_ERROR)
   {
      XEmac_Reset((XEmac *)CallBackRef);                                    // Reset EMAC

      (void)XEmac_SetMacAddress((XEmac *)CallBackRef, LocalAddress);        // SET MAC ADDRESS
      (void)XEmac_SetOptions((XEmac *)CallBackRef, TEST_1_OPTIONS);         // SET OPTIONS

      (void)XEmac_Start((XEmac *)CallBackRef);                              // Start EMAC
   }
        xil_printf("Exiting FIFO Error Handler\n\r");
}
```

Sniffer_GlobalVariables.h

```
/*****************************************************************************
*
*       Sniffer_GlobalVariables.h
*       Simple Network Sniffer __ Global Variables
*
*       NOTE:The local MAC Address of the sniffer is AA-AA-AA-AA-AA-AA
*               (randomly chosen)
*               This MAC Address is reserved and should not be used for another
*               Node on the network.
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher-John Comis (University of Toronto)
*       at the following email address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/



#ifndef SNIFFER_GLOBALVARIABLES_H
#define SNIFFER_GLOBALVARIABLES_H
```

```
/**************************** Constants Definitions ***********************************/

#define XEM_MAX_FRAME_SIZE_IN_WORDS ((XEM_MAX_FRAME_SIZE / sizeof(Xuint32)) + 1)

#define TEST_1_OPTIONS          (XEM_UNICAST_OPTION | XEM_INSERT_PAD_OPTION | \
                                 XEM_INSERT_FCS_OPTION | XEM_INSERT_ADDR_OPTION | \
                                 XEM_OVWRT_ADDR_OPTION | XEM_PROMISC_OPTION)

#define LED   0x1                               // bit 0 of GPIO is connected to the LED

#define INTC_INPUT_EMAC             0           // EMAC has higher priority (check add/edit cores)

#define INTC_DEVICE_ID          0


#define ACK_L2_PAYLOAD_SIZE     36
#define ACK_L1_PAYLOAD_SIZE     46
#define ACK_FRAME_SIZE          60
#define L2_HEADER_SIZE          10

#define ACK_MESSAGE_TYPE                1
#define NON_FRAGMENTED_MESSAGE_TYPE     0
#define FIRST_FRAGMENT_TYPE             2
#define INTERNAL_FRAGMENT_TYPE          3
#define LAST_FRAGMENT_TYPE              4
#define MAX_L2_PAYLOAD                  1490


/*************************  Instance Definitions  ************************************/
```

```
static XGpio Gpio;
static XEmac Emac;
static XIntc InterruptController;

XEmac *EmacPtr;                                         // Initialized in Init.h

/*************************  MAC Addresses Definitions    ******************************/

static Xuint8 SnifferAddress[XEM_MAC_ADDR_SIZE] = //   48 bits Sniffer address
{
   0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA
};

Xuint8 * LocalAddress = SnifferAddress;

/*************************  Buffer Variables    ******************************/

static Xuint32 RecvBuffer[XEM_MAX_FRAME_SIZE_IN_WORDS];              // holds incoming data
static Xuint32 FrameLen;
static Xuint32 TemporaryBuffer;

static Xuint8 DegreeofInformation;   // this will be set by the Gpio DIP Switch to select whether to get full info
                                                                    // or just type of frame and source and destination
static Xuint32 debouncing;

/*************************  Global Variables    ******************************/

char array[17];                                 // used to hold MAC addresses in printable standard format
```

```
/************************* Acknowledgement Variables **********************/

XEmac_Stats Stats;  // One Variable for the EMAC Module.

#endif
```

Sniffer_DebugFunctions.h

```
/*****************************************************************************
*
*       Sniffer_DebugFunctions.h
*       Simple Network Sniffer __ Debug Tools
*
*       This contains some functions used to parse the frame and
*       display the various fields in the frame
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher-John Comis (University of Toronto)
*   at the following email address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/
```

```
#ifndef SNIFFER_DEBUGFUNCTIONS_H
#define SNIFFER_DEBUGFUNCTIONS_H

#include "Sniffer_GlobalVariables.h"

/*
*   returns the character equivalent of an integer
*/
char map(int x);


/*
* Pass it a Pointer to the start of the Ethernet Header
* Returns the Layer 2 Type Field
*/
Xuint16 GetType(Xuint8 *MsgBuffer);


/*
* Pass it a Pointer to the start of the Ethernet Header
* Returns the Layer 2 Sequence Number Field
*/
Xuint32 GetSeqNum(Xuint8 *MsgBuffer);

/*
* Pass it a Pointer to the start of the Ethernet Header
* Returns the Layer 2 Message Size Field
*/
Xuint32 GetSize(Xuint8 *MsgBuffer);
```

```
/*
* Pass it a pointer to the EMAC Address
* Fills the global variable 'array' with
* the ETHERNET Address in standard format
*/
void fill_array(Xuint8 *Ptr);


/*
* Pass it a pointer to the Ethernet Frame
*
* Prints the Layer 2 Variables
*
* Message Header Format:     Type                    _ 2 bytes
*                                                    Sequence Number       _ 4 bytes
*                       Message Length      _ 4 bytes
*/
void Print_Message_Fields(Xuint8 *MsgBuffer, int MsgLen, Xuint8 Extra);

/*
* Pass it a pointer to the Ethernet frame
*
* and the MsgLen as returned by the Receive function
*
* Ethernet Header Format:     Destination Address _ 6 bytes
*                                                    Source Address      _ 6 bytes
*                       Len / Type Field    _ 2 bytes
*
```

```
*/
void Print_Frame_Fields(Xuint8 *MsgBuffer, int MsgLen, Xuint8 Extra);

#endif
```

Sniffer_DebugFunctions.c

```
/*****************************************************************************
*
*       Sniffer_Handlers.c
*       Simple Network Sniffer __ Debug Tools
*
*       This contains some functions used to parse the frame and
*       display the various fields in the frame
*
*       Author:         Patrick E. Akl
*                       American University of Beirut
*                       August 2004
*
*       For any comments, suggestions or bug discovery, please contact me at the following
*       email address : patrickakl@hotmail.com
*
*       or contact Christopher-John Comis (University of Toronto)
*       at the following email address:
*                                       comis@eecg.toronto.edu
*
*****************************************************************************/

#include "Sniffer_GlobalVariables.h"
```

```
/*
*   returns the character equivalent of an integer
*/
char map(int x)
{
        switch(x)
        {
        case 0:
                return '0';
        case 1:
                return '1';
        case 2:
                return '2';
        case 3:
                return '3';
        case 4:
                return '4';
        case 5:
                return '5';
        case 6:
                return '6';
        case 7:
                return '7';
        case 8:
                return '8';
        case 9:
                return '9';
        case 10:
                return 'A';
```

```
		case 11:
			return 'B';
		case 12:
			return 'C';
		case 13:
			return 'D';
		case 14:
			return 'E';
		case 15:
			return 'F';
		}
}


/*
* Pass it a Pointer to the start of the Ethernet Header
* Returns the Layer 2 Type Field
*/
Xuint16 GetType(Xuint8 *MsgBuffer)
{
		Xuint8 * Ptr = MsgBuffer;
		Ptr+=14;
		return *(Xuint16 *)Ptr;
}

/*
* Pass it a Pointer to the start of the Ethernet Header
* Returns the Layer 2 Sequence Number Field
*/
Xuint32 GetSeqNum(Xuint8 *MsgBuffer)
```

```
{
        Xuint8 * Ptr = MsgBuffer;
        Ptr+=16;
        return *(Xuint32 *)Ptr;
}

/*
* Pass it a Pointer to the start of the Ethernet Header
* Returns the Layer 2 Message Size Field
*/
Xuint32 GetSize(Xuint8 *MsgBuffer)
{
        Xuint8 * Ptr = MsgBuffer;
        Ptr+=20;
        return *(Xuint32 *)Ptr;
}


/*
* Pass it a pointer to the EMAC Address
* Fills the global variable 'array' with
* the ETHERNET Address in standard format
*/
void fill_array(Xuint8 *Ptr)
{
        Xuint8 mycounter;
        for(mycounter = 0 ; mycounter < 17 ; mycounter += 3)
        {
                array[mycounter+1] = map(*(Ptr) & 0xF);
                array[mycounter] = map( ( *(Ptr) & 0xF0 ) >> 4);
```

```
                Ptr++;
                if(mycounter<15) array[mycounter+2] = '-';
        }
}


/*
* Pass it a pointer to the Ethernet Frame
*
* Prints the Layer 2 Variables
*
* Message Header Format:    Type                    _ 2 bytes
*                                                   Sequence Number     _ 4 bytes
*                       Message Length     _ 4 bytes
*/
void Print_Message_Fields(Xuint8 *MsgBuffer, int MsgLen, Xuint8 Extra)
{
        Xuint32 mycounter;
        Xuint32 LenField;
        Xuint16 TypeofMess;
        Xuint32 FragmentPayloadLength; // since FragmentLength can be different
                                                    // than Message PayloadLength
        Xuint8 *MesgPtr = MsgBuffer;
        Xuint8 *TemporaryPtr = (Xuint8 *)TemporaryBuffer;

        if(Extra == 1)
        {
                xil_printf("Message Parameters\r\n");
                xil_printf("____________________\r\n");
        }
```

```
MesgPtr+=14;
TypeofMess = *(Xuint16 *)MesgPtr;
xil_printf("Message Type Field                  : %d\n\r", TypeofMess);

MesgPtr+=2;
if(Extra == 1)
{
        xil_printf("Message Sequence Nb Field       : %d\n\r", *(Xuint32 *)MesgPtr);
}

MesgPtr+=4;

LenField = *(Xuint32 *)MesgPtr;

xil_printf("Message Length                     : %d\n\r", LenField);

MesgPtr+=4;

if(LenField <= 1490)
        FragmentPayloadLength = LenField;          // The Frame contains the Message header (10 bytes)
                                                                                // and the whole message itself as its payload
          // case of fragmented message
        else
        {
                if(TypeofMess == FIRST_FRAGMENT_TYPE || TypeofMess == INTERNAL_FRAGMENT_TYPE) // frame is full
                        FragmentPayloadLength = MAX_L2_PAYLOAD;
                else if (TypeofMess == LAST_FRAGMENT_TYPE) // last fragment of a message
                {
                        if(LenField % MAX_L2_PAYLOAD == 0) // 1490 payload size of last fragment
```

```
                        FragmentPayloadLength = MAX_L2_PAYLOAD;
                else
                        FragmentPayloadLength = (LenField % MAX_L2_PAYLOAD);
        }
}

if(Extra == 1)
{
        // Printing the first byte of the fragment payload
                xil_printf("Fragment Payload 1st byte: %x\n\r", *(Xuint8 *)MesgPtr);

        // Getting to the last byte of the payload
                MesgPtr += (FragmentPayloadLength - 1);

        // Printing the last byte of the fragment payload
                xil_printf("Fragment Payload last byte: %x\n\r", *(Xuint8 *)MesgPtr);

        /*
                xil_printf("Frame Payload                   :\r\n");
                for(mycounter=0 ; mycounter < FragmentPayloadLength ; mycounter++)
                {
                        xil_printf("%x", *(Xuint8 *)MesgPtr);
                        MesgPtr++;
                }

        */
}

if(TypeofMess == 1)             // ack
        xil_printf("\r\n_____________________________________________________\r\n");
```

```
}

/*
* Pass it a pointer to the Ethernet frame
*
* and the MsgLen as returned by the Receive function
*
* Ethernet Header Format:     Destination Address _ 6 bytes
*                                                 Source Address      _ 6 bytes
*                         Len / Type Field    _ 2 bytes
*
*/
void Print_Frame_Fields(Xuint8 *MsgBuffer, int MsgLen, Xuint8 Extra)
{
        Xuint8 *MesgPtr = MsgBuffer;

        if(Extra == 1)
        {
        xil_printf("Frame Parameters\r\n");
        xil_printf("_____________________\r\n");

        xil_printf("Frame Length                    : %d\r\n", MsgLen);
        }

        fill_array(MesgPtr);
        xil_printf("Destination MAC                 : %s\r\n", array);

        MesgPtr+=6;

        fill_array(MesgPtr);
```

```
        xil_printf("Source      MAC                     : %s\r\n", array);

        if(Extra == 1)
        {
                MesgPtr+=6;
                xil_printf("Payload Length                  : %d\r\n", *(Xuint16 *)MesgPtr);
        }
}
```

Future Work

Due to time constraints, some engineering decisions involved schemes that are known not to be the most efficient and the most suitable for the project, in fact the optimum decisions would need more time for implementation and testing.

1. Implementing a more efficient Acknowledgement and Retransmission Scheme

The simplest way possible for an acknowledgment scheme is to have an acknowledge frame sent for each frame received. However, this implementation is clearly inefficient since the overhead introduced by acknowledge frames approaches 50 %. This is, however, a very easy scheme to implement regarding the sequence numbering scheme and retransmission schedule. A more efficient scheme, yet harder and more time consuming to implement, would make use of adaptive sliding window schemes or any other scheme that is actually used in various complex reliable protocols such as TCP (*Transfer Control Protocol*).

2. Implementing a more efficient Sniffer node

The sniffer node implemented in the project is a simple networking node configured in promiscuous mode. It parses the Ethernet header fields as well as the higher layer protocol fields and displays them on the hyperterminal screen on the computer that reads data from the serial port. The user selects with a DIP Switch whether to print all the fields or some subset of it. These were called heavy or light monitoring.

This implementation of a sniffer node is only for demonstration and suffers from the following problems:

- At very high frame rates exchanged by the nodes on the network, the sniffer node would not be able to track all the frames because its speed is limited by the speed of the serial port that transfers data to the HyperTerminal screen at 9600 bps. So, at high frame rates, the data displayed on the screen would not be consistent with the actual frame parameters because the FIFO queues of the Ethernet controller would be overflowing. Moreover, the user would not be able to read all the data being continuously displayed on the screen and the system is memoryless.
- The Sniffer node would only work when connected on a hub because on a switch, all the frames would be directed to their destination and would not reach the sniffer, whereas the hub is a simple repeater which will retransmit the frame to all the nodes connected to it.

It would be nice and very easy to modify this sniffer to obtain a more efficient network monitoring system that would be very useful in large scale and system with high statistical average bitrates:

- Keeping a log of the statistics instead of displaying them would help us keep track of many parameters efficiently. To achieve this, we would need to have some data structure tables indexed by destination and / or source MAC Addresses.

Main Sources

1. Embedded Design Kit training at the University of Toronto. (set of PDF files)
   - *MicroBlaze Reference Guide.*
   - *Multimedia Schematics.*
   - *Multimedia User Guide.*
   - *Processor IP User Guide.*
   - *Multimedia Training File.*
   - *Platform Studio User Guide.*
   - *User Core Template User Guide.*
   - *VERILOG Tutorial.*
   - *Xilinx Drivers.*

2. ECE532_*Digital Hardware* at the University of Toronto (Lab experiments)

3. Handout on VERILOG HDL_*Daniel C. Hyde*_Bucknell University.

4. VERILOG Tutorial_*Deepak Kumar Tala*_www.deeps.com.

5. www.xilinx.com.