

**Department of Electrical and Computer Engineering**

#### **EGC455- System-On-Chip**

**Final Project**

**Small Scale Layered Networking Device for an IPv4**

|  |  |
| --- | --- |
| **Date:12/19/2016** | **Fall Semester 2016** |

|  |  |  |
| --- | --- | --- |
| **Group Members** | **Department** | **Major Contribution** |
| **Joseph Abel** | **CE** | **Report** |

**Lecture Instructor: Professor Mike Otis**

**Abstract:**

The objective of this project is to implement and design a small scale layered networking device that interfaces more than one hardware device in order to show transmission and receiving serial data packets over the wire. We then use a CRC4 mechanism to verify the packet validity.

**Table Of Contents**

**Introduction 3**

**Theory 3-6**

**Figure 1 HDLC Frame 4**

**Figure 2 Transceiver block diagram 5**

**Design 6-9**

**Figure 3: DE0-NANO-SOC FPGA Board 7**

**Figure 4: Receiving Module 8**

**Figure 5: Sending Functionality 9**

**Results 10-11**

**Figure 4: Tying all the Hardware Together 10**

**Conclusion 11**

**Works Cited 12**

**Appendix A: Verilog Receiving Code 13-18**

**Module SOCFinal 13-14**

**Module SeqDet 14-15**

**Module dataAddr 15-16**

**Module dataCap 17**

**Module CRC4 18**

**Appendix B: C Sending Code 19-22**

**Main.c 19-22**

**Introduction:**

Layered networking has consisted of a tie between a hardware layer and software layer of devices for some time. In this project, we will implement a small version of that. Using our Terasic board from Altera, we will receive data from our Texas Instruments board. We will demonstrate a couple types of transmission. Using a 2 bit bus, we will demonstrate unicast transmission and a multicast transmission. In order to demonstrate a fully working model of this on a receiving device, we will use two receiving devices. The difference in these two devices will be their unicast address. So when we are unicasting, the device with the respective unicast address will receive those bits and be represented by the CRC mechanism and our acknowledgement bits.

**Theory:**

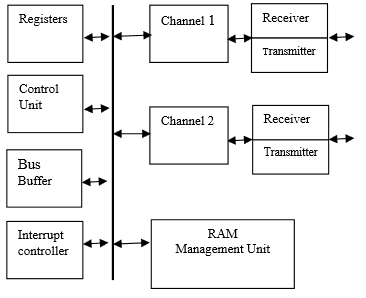
The first article we used on this project is called the “Implementation of HDLC Protocol Using Verilog”. This article explained about how High-Level Data Link Control protocol (HDLC) is a vital concept when you are dealing with the data link layer. This HDLC was established by the International Organization for Standardization (ISO) which is widely used in digital communications and are the bases of many other data link control protocols. The HDLC protocol is comprised of the flag which has the binary sequence of 01111110. This technique of inserting these bits are called bit stuffing and this is how each frame begins in the sequence, the next field used in the HDLC protocol is the addressing field which is a programmable size of a single octet. This field can contain the value programmed into the transmit address, the next field is the control field which is how you control the communication process from the field with different commands, responses and sequences numbers and is used to maintain the data flow accountability of the link. In the HDLC procedure there are also two different modules the transmit side and the receive side. The transmit side is the data interface which provides a byte-wide interface between the transmission host and the HDLC controller. The transmit data is loaded into the controller on the rising edge of the clock when the write input is asserted. The receive module accepts a bit stream on which ever master you have in your system. In typical HDLC protocol mode all receiver frames are presented to the host on the output register.[1]. The second article is called the “Design and Implementation of HDLC Controller by Using Crc-16“. This article shows how the group of protocols for transmitting synchronous data packets between point to point nodes operates. The HDLC protocol is used frequently in high speed hardware implementation for new technologies and innovative thoughts in order to check validity and possible advantage is possible by using FPGA. The HDLC frame format is as follows in the figure below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| FLAG | ADDRESS | DATA | FCS | FLAG |

**Figure 1:** HDLC frame format

There is also another technique called transparency which is performed by the core on all bits between the FCS. This operation involves inserting a zero after any sequence of 5 consecutive ones in the transmitted data stream this is also called bit stuffing for short. The HDLC layer can also be done alongside the CRC calculation this is done by modulo 2 division.[2] , The third paper called “Implementation of HDLC controller Design Using Verilog HDL” was used in order to help us understanding the OSI model which makes it clear to us that we are using the layer 2 and layer 3 (the data link layer and the physical layer). This model helped show us how these layers interconnect and form a full functioning network protocol. The physical layer specifies how the bits are transmitted and this can include the data and signal encoding from the connector. This article also goes over the master-slave interaction with the Data Link layer which is exactly how we implemented this project. This is done by establishing a point to point network topology and establishing a Wide Area network (WAN) links. This layer also shows us how the two entities of communication at any time called Normal Response Mode (NRM, also known as Unbalanced), and a peer to peer relationship call Asynchronous Balanced Mode. NRM has all connection

Initialization, configuration and control done by the designated master. The fourth article is called the “Optimization of multi channel HDLC protocol transceiver using Verilog” this article showed us how the actual transceiver block diagram would be made. This is shown in the following figure below:

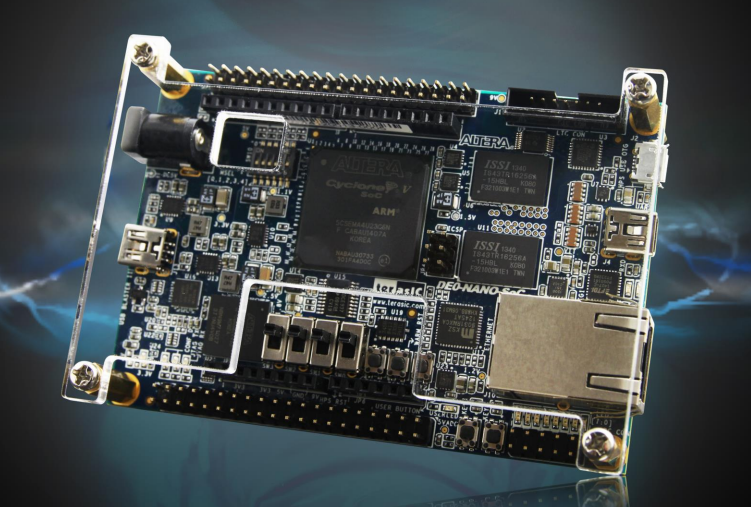


**Figure 2**: Transceiver block diagram

This transceiver block diagram has the control unit and registers, the transmitter, the receiver, dual port for RAM, RAMS management and an interrupt controller all along a bus of different channels receiving data. The interrupt controller is used when transmitting or receiving the interrupt status includes if the operation has finished successfully or is going to throw an error. This is used for instance if there is an error status such as an FCS error or frame length. When all 1’s or previous bits of frame is not properly sent due to some external error between the transmitter or receiver this interrupt will occur. This project also shows how the HDLC protocol transceiver and receiver side maintain a full duplex of channels across communication channels.[4] The fifth article used is called “Next Generation Internet using the IPv4 address Exhaustion, Mitigation strategies and implications for the U.S. this article helped us show the growth of the IPv4 to IPv6 the newer protocol. This article also gave a hierarchical view of what the system actually is applicable towards such as developing a network router to solve the routing scale problem. This is done by breaking each IP address into a network number and a host number on that network. The networks vary greatly in size. The IPv4 introduced network classes to classify the finite composition of the internet. The IPv4 also has a greater scale with in the United States since the IPv4 addressing scheme has come the IPv6 deployment has been delayed which means the United States risks becoming an island in the global next generation Internet. [5]

**Design:**

First, we designed our receive side. In order to implement the receive side of our unit, we will make use of an ARM powered FPGA board from Terasic, the DE0-NANO-SOC board as seen below in figure 1. The project will later consist of of two pieces of hardware to handle their respective functionalities, but for testing and verification purposes, we used a switch for our data bit and we used a button for our clock on our FPGA board. We designed four modules to handle 4 parts of our receive side. We will first detect a sequence. The sequence will allow for receiving data to an address. The next data in will be the data address where the data is going. From this sequence, the device will decide whether the incoming data is intended for that party. If it is intended, the following module will store that data. Once the data is stored, it will undergo a CRC4 check in order to verify its validity.

  
**Figure 3: DE0-NANO-SOC FPGA Board**

We first implemented a state machine which took on the role of a sequence detector. Basically, the sequence detector would take in bit after bit until it’s 8 bit sequence ‘01111110’ is satisfied. It then has an output acknowledgement bit that represents that the sequence has been satisfied. It will continue doing this until a call for reset tells it to start looking for it’s sequence again.

Now, another state machine was implemented in order to decide whether data transmission would occur. This state machine was essentially another sequence detector however in this case, this sequence detector would be pointing to a specific data address in which we would be writing to. So, if within 8 clock cycles, the correct sequence of 8 bits, the data address was satisfied, we would receive data into said address. This sequence detector also handled multicast. Multicast means that the data will be written to all addresses. This was represented by the sequence ‘1XXXXXXX’ where the X’s represent don’t cares. So in other words, 1 in the MSB for data address meant that a multicast was occurring, so after 8 clock cycles, the unit would definitively receive the data.

From there, a module would receive 8 bits of data, and then it would reset the sequence detector and the data addressing module using an acknowledge bit. From there, the data is sent to the CRC module.

The upon satisfaction of the data being captured, the CRC module will take the data that was received and run it through a CRC4 mechanism as a verification procedure. This CRC will re-occur after every data capture and be displayed on LEDs.

All of these modules were enabled by satisfaction of the previous module. So it happens like this.



**Figure 4: Receiving Module**

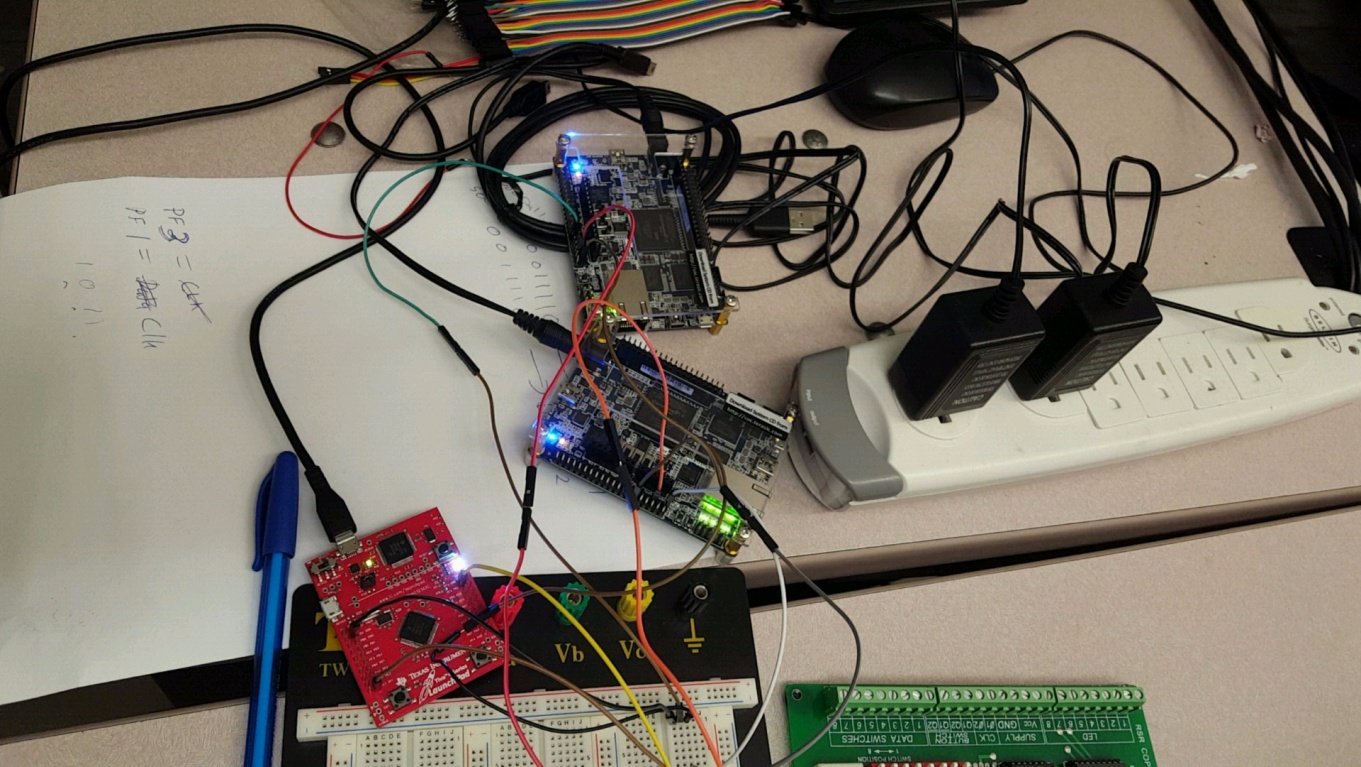
We were able to process data through as expected by using the switches and push buttons as our clock and data.

The send side of our module was handled using C code and the Texas Instruments TM4C evaluation board. We will use two pins on the board as digital outputs. One pin will just just toggle a clock and one pin will be used for data. We made a few different C functions. We first made a clock function which basically functioned as a frequency divider. In this case, in order to lower the chances of electromagnetic interference, we slowed down the clock in order to make sure we captured data accurately. The clock function was called amongst toggling the data bit as needed.

We also had two unicast functions. The need for two was to demonstrate that two receiving devices would receive their respective data correctly when called upon. So, each unicast function mapped to a separate address, and then the receiving end would be responsible for capturing the data when they are called upon. There is also a multicast function in which every prepared receiver would receive data. The sending was very brute force, however, the sequence can be seen in figure 3 below.

  
**Figure 5: Sending Functionality**

**Results**

After all is designed, we will tie everything together on the hardware level. Below, you will see two terasic boards and a TM4C board all tied together. 

**Figure 6: Tying all the Hardware Together**

On the Terasic board, 7 LEDs were used. 4 were used to display the output of the CRC mechanism. The other 3 showed the stages as data was processed through. For instance, after the original sequence detector was satisfied, LED[0] would light. After that, the dataaddr module was enabled. If that was satisfied, LED[1] should light. If not, LED[0] would turn off indicating that it was not satisfied, and we are back to stage 1. If it was satisfied, the datacap stage would occur. After the eight bits are captured, LED[2] is lit and LED 0 and 1 go out indicating that the full sequence was complete. Once the sequence is complete, CRC4 generates its value and the sequence detector is ready for round two.

The only real difference between these two Terasic boards is the data address they use for unicasting. During verification, we used our C code in order to clock in data. We tested three different cases. Unicast1, unicast2, and multicast. We see while executing unicasts, that the unicast unit we are not casting to gets satisfied and the sequence detector stage, but eight bits later after data addressing, it begins looking for the original sequence again because it was unable to reach the data capture stage. The unit that is being unicasted to then continues to the data capture stage. Data is captured and then sent to CRC. In the case of our program, unicast1 casted to device 1 at data address ‘00011110’, and unicast 2 casted to device 2 at data address ‘00111100’.

**Conclusion**

By the end of this project, we were able to demonstrate a small scale networking system. We conducted data and clock over a serial bus and showed the whole receiving process on terasic board end. Overall, in the end, the only real hard thing about this project was understanding what was being asked. It took a few times of being explained the concept that we were to display to get the final product, which worked exactly as expected.

**Works Cited**

[1]"I m p l e m e n t a t i o n o f H D L C P r o t o c o l U s i n g V e r i l o g", *Ijraset.com*, 2016. [Online]. Available: http://www.ijraset.com/fileserve.php?FID=968. [Accessed: 20- Dec- 2016].

[2] "Design and Implementation of HDLC Controller by Using Crc - 16", *http://www.ijmer.com/papers/Vol3\_Issue1/AC311218.pdf*, 2016. [Online]. Available: http://www.ijmer.com/papers/Vol3\_Issue1/AC311218.pdf. [Accessed: 20- Dec- 2016].

[3]"Implementation of HDLC Controller Design using Verilog HDL", *https://www.researchgate.net/profile/Armaan\_Nagpurwala/publication/260872170\_Implementation\_of\_HDLC\_Controller\_Design\_using\_Verilog\_HDL/links/0a85e53292eb1bb2e5000000.pdf*, 2016. [Online]. Available: https://www.researchgate.net/profile/Armaan\_Nagpurwala/publication/260872170\_Implementation\_of\_HDLC\_Controller\_Design\_using\_Verilog\_HDL/links/0a85e53292eb1bb2e5000000.pdf. [Accessed: 20- Dec- 2016].

[4]"Optimization of multi - channel HDLC protocol transceiver using Verilog", *http://www.ijcsi.org/papers/IJCSI-9-2-2-259-262.pdf*, 2016. [Online]. Available: http://www.ijcsi.org/papers/IJCSI-9-2-2-259-262.pdf. [Accessed: 20- Dec- 2016].

[5]"Next Generation Internet: IPv4 Address Exhaustion, Mitigation Strategies and Implications for the U.S.", *https://www.ieeeusa.org/policy/whitepapers/IEEEUSAWP-IPv62009.pdf*, 2016. [Online]. Available: https://www.ieeeusa.org/policy/whitepapers/IEEEUSAWP-IPv62009.pdf. [Accessed: 20- Dec- 2016].

**Appendix A:** Verilog Code

**Verilog**

module SOCFinal(in, clk, rst, SeqSat, dataSat, captured, CRCOut);

input in, clk, rst;

wire call\_for\_rst, done, captured;

wire [7:0] data;

output wire SeqSat, dataSat, captured;

output wire [3:0] CRCOut;

assign reset = ~rst | call\_for\_rst | done;

SeqDet sd(.in(in), .clk(clk), .rst(reset), .sat(SeqSat));

dataAddr ad(.in(in), .clk(clk), .rst(reset), .prev\_sat(SeqSat), .sat(dataSat), .return(call\_for\_rst));

dataCap dc(.in(in), .clk(clk), .prevSat(dataSat), .return(done), .data(data), .captured(captured));

CRC4 CRC(.a(data), .clk(done), .CRC(CRCOut));

endmodule

module SeqDet(in, clk, rst, sat);

input in, clk, rst;

output reg sat;

reg[3:0] state;

always@(posedge clk or posedge rst)

begin

if (rst)

begin

state = 4'b0000;

sat = 0;

end

else

case (state)

4'b0000:

if(~in)

state = 4'b0001;

4'b0001:

if(in)

state = 4'b0010;

else

state = 4'b0001;

4'b0010:

if(in)

state = 4'b0011;

else

state = 4'b0001;

4'b0011:

if(in)

state = 4'b0100;

else

state = 4'b0001;

4'b0100:

if(in)

state = 4'b0101;

else

state = 4'b0001;

4'b0101:

if(in)

state = 4'b0110;

else

state = 4'b0001;

4'b0110:

if(in)

state = 4'b0111;

else

state = 4'b0001;

4'b0111:

if(~in)

begin

state = 4'b1000;

sat = 1;

end

else

state = 4'b0000;

4'b1000:

if(sat)

state = 4'b1000;

else

begin

state = 4'b0000;

sat = 0;

end

endcase

end

endmodule

module dataAddr(in, clk, rst, prev\_sat, sat, return); //This module’s data addr seq was

altered for unicasting to two devices

input in, clk, rst, prev\_sat;

output reg sat, return;

integer count;

reg[3:0] state;

always@(posedge clk or posedge rst)

begin

if (rst)

begin

state = 4'b0000;

sat = 0;

count = 0;

return = 0;

end

else

if(prev\_sat)

begin

count = count + 1;

case (state)

4'b0000:

if(in) //multicast

state = 4'b1111;

else

state = 4'b0001;

4'b0001:

if(~in)

state = 4'b0010;

4'b0010:

if(~in)

state = 4'b0011;

4'b0011:

if(in)

state = 4'b0100;

4'b0100:

if(in)

state = 4'b0101;

4'b0101:

if(in)

state = 4'b0110;

4'b0110:

if(in)

state = 4'b0111;

4'b0111:

if(~in)

begin

state = 4'b1000;

sat = 1;

end

4'b1000:

if(sat)

state = 4'b1000;

else

begin

state = 4'b0000;

sat = 0;

end

4'b1111:

if(count == 8)

begin

state = 4'b1000;

sat = 1;

end

endcase

if(count == 8)

if(~sat)

return = 1;

end

end

endmodule

module dataCap(in, clk, prevSat, return, captured, data);

input in, clk, prevSat;

output reg return, captured;

output reg [7:0] data;

integer count, i;

always@(posedge clk)

begin

if(~prevSat)

begin

count = 0;

end

if(prevSat)

begin

if(count < 8)

data[count] = in;

count = count + 1;

end

end

always@(negedge clk)

begin

if(count == 7)

begin

return = 1;

captured = 1;

end

else

begin

return = 0;

captured = 0;

end

end

endmodule

module CRC4(a, clk, CRC);

input clk;

input [7:0] a;

reg b;

output reg [3:0] CRC;

integer i;

always@(posedge clk)

begin

CRC = 4'b0000;

#200;

for(i = 7; i >= 0; i = i - 1)

begin

b = a[i] ^ CRC[3];

CRC[3] = CRC[2];

CRC[2] = CRC[1];

CRC[1] = CRC[0] ^ b;

CRC[0] = b;

end

for(i = 4; i > 0; i = i - 1)

begin

b = 1'b0 ^ CRC[3];

CRC[3] = CRC[2];

CRC[2] = CRC[1];

CRC[1] = CRC[0] ^ b;

CRC[0] = b;

end

end

endmodule

**Appendix B: C Sending Code**

// 0.Documentation Section

// main.c

// Runs on LM4F120 or TM4C123

// C6\_InputOutput, Input from PF4, output to PF2 (blue LED)

// Authors: Daniel Valvano, Jonathan Valvano and Ramesh Yerraballi

// Date: July 8, 2013

// LaunchPad built-in hardware

// SW1 left switch is negative logic PF4 on the Launchpad

// SW2 right switch is negative logic PF0 on the Launchpad

// red LED connected to PF1 on the Launchpad

// blue LED connected to PF2 on the Launchpad

// green LED connected to PF3 on the Launchpad

// 1. Pre-processor Directives Section

// Constant declarations to access port registers using

// symbolic names instead of addresses

#define GPIO\_PORTF\_DATA\_R (\*((volatile unsigned long \*)0x400253FC))

#define GPIO\_PORTF\_DIR\_R (\*((volatile unsigned long \*)0x40025400))

#define GPIO\_PORTF\_AFSEL\_R (\*((volatile unsigned long \*)0x40025420))

#define GPIO\_PORTF\_PUR\_R (\*((volatile unsigned long \*)0x40025510))

#define GPIO\_PORTF\_DEN\_R (\*((volatile unsigned long \*)0x4002551C))

#define GPIO\_PORTF\_LOCK\_R (\*((volatile unsigned long \*)0x40025520))

#define GPIO\_PORTF\_CR\_R (\*((volatile unsigned long \*)0x40025524))

#define GPIO\_PORTF\_AMSEL\_R (\*((volatile unsigned long \*)0x40025528))

#define GPIO\_PORTF\_PCTL\_R (\*((volatile unsigned long \*)0x4002552C))

#define PF4 (\*((volatile unsigned long \*)0x40025040))

#define PF3 (\*((volatile unsigned long \*)0x40025020))

#define PF2 (\*((volatile unsigned long \*)0x40025010))

#define PF1 (\*((volatile unsigned long \*)0x40025008))

#define PF0 (\*((volatile unsigned long \*)0x40025004))

#define GPIO\_PORTF\_DR2R\_R (\*((volatile unsigned long \*)0x40025500))

#define GPIO\_PORTF\_DR4R\_R (\*((volatile unsigned long \*)0x40025504))

#define GPIO\_PORTF\_DR8R\_R (\*((volatile unsigned long \*)0x40025508))

#define GPIO\_LOCK\_KEY 0x4C4F434B // Unlocks the GPIO\_CR register

#define SYSCTL\_RCGC2\_R (\*((volatile unsigned long \*)0x400FE108))

// 2. Declarations Section

// Global Variables

unsigned long In; // input from PF4

unsigned long Out; // output to PF2 (blue LED)

// Function Prototypes

void PortF\_Init(void);

void clock(void);

void multicast(void);

void unicast1(void);

void unicast2(void);

void data(void);

void sequence(void);

// 3. Subroutines Section

// MAIN: Mandatory for a C Program to be executable

int main(void){ // initialize PF0 and PF4 and make them inputs

PortF\_Init(); // make PF3-1 out (PF3-1 built-in LEDs)

sequence();

multicast();

data();

}

void sequence(void)

{

GPIO\_PORTF\_DATA\_R = 0xF7; //DATA = 0

clock();

clock();

clock();

GPIO\_PORTF\_DATA\_R |= 0x08; //DATA = 1

clock();

clock();

clock();

clock();

clock();

clock();

GPIO\_PORTF\_DATA\_R &= 0xF7; //DATA = 0

clock();

}

void multicast(void)

{

GPIO\_PORTF\_DATA\_R |= 0x08;

clock();

clock();

clock();

clock();

clock();

clock();

clock();

clock();

}

void unicast1(void)

{

GPIO\_PORTF\_DATA\_R &= 0xF7;

clock();

clock();

GPIO\_PORTF\_DATA\_R |= 0x08;

clock();

clock();

clock();

clock();

GPIO\_PORTF\_DATA\_R &= 0xF7;

clock();

clock();

}

void unicast2(void)

{

GPIO\_PORTF\_DATA\_R &= 0xF7;

clock();

clock();

clock();

GPIO\_PORTF\_DATA\_R |= 0x08;

clock();

clock();

clock();

clock();

GPIO\_PORTF\_DATA\_R &= 0xF7;

clock();

}

void data(void)

{

GPIO\_PORTF\_DATA\_R |= 0x08;

clock();

clock();

clock();

clock();

clock();

clock();

clock();

clock();

}

void clock(void)

{

int i;

GPIO\_PORTF\_DATA\_R &= 0xFD;

for(i = 0; i < 100000; i++);

GPIO\_PORTF\_DATA\_R |= 0x02;

for(i = 0; i < 100000; i++);

}

// Subroutine to initialize port F pins for input and output

// PF4 is input SW1 and PF2 is output Blue LED

// Inputs: None

// Outputs: None

// Notes: ...

void PortF\_Init(void){ volatile unsigned long delay;

SYSCTL\_RCGC2\_R |= 0x00000020; // 1) activate clock for Port F

delay = SYSCTL\_RCGC2\_R; // allow time for clock to start

GPIO\_PORTF\_LOCK\_R = 0x4C4F434B; // 2) unlock GPIO Port F

GPIO\_PORTF\_CR\_R = 0x1F; // allow changes to PF4-0

// only PF0 needs to be unlocked, other bits can't be locked

GPIO\_PORTF\_AMSEL\_R = 0x00; // 3) disable analog on PF

GPIO\_PORTF\_PCTL\_R = 0x00000000; // 4) PCTL GPIO on PF4-0

GPIO\_PORTF\_DIR\_R = 0x0E; // 5) PF4,PF0 in, PF3-1 out

GPIO\_PORTF\_AFSEL\_R = 0x00; // 6) disable alt funct on PF7-0

GPIO\_PORTF\_PUR\_R = 0x11; // enable pull-up on PF0 and PF4

GPIO\_PORTF\_DEN\_R = 0x1F; // 7) enable digital I/O on PF4-0

}