Data Communications
EGC 412
Final Project

Packet Communication

Team Members:	**Contributions:**
Mitchell Wagner	Theory, Code, Implementation, Write Up
Paul Boston	Theory, Code, Implementation, Write Up

Professor: Mike Otis

Abstract:

All computers need to be able to transfer information to different devices. The main issue in sending data is receiving the correct data that was sent. This issue maybe incorrect from the receiver not being able to distinguish between data and other information. Also where the information needs to be sent to if there are multiple units that want the same information. Finally another major portion is when is that data finished. All these situations are required in creating an accurate data communication network.

Table of Contents

Introduction:

Data Communication is the key aspect to any type of computer software. Data needs to be sent to and received from these computer systems all the time. It's very important to have the correct data be sent and for the other device to recognize that it has received that data correctly in the right location. Key aspects in receiving correct data are: Preamble, Addressing and CRC.

Preamble is a portion of the transmitted data that is used to check that everything is lined up so that the processor knows at what point of location it is in. Addressing is used to inform the processor where to send the data that it will be passed. The CRC is used after the data to inform the receiving side that it has finished receiving data. A flowchart depicting each aspect is represented in Figure 1, Preamble is red, Addressing is blue and Data and CRC are green. Each of these components help to make a reliable communications between different devices.

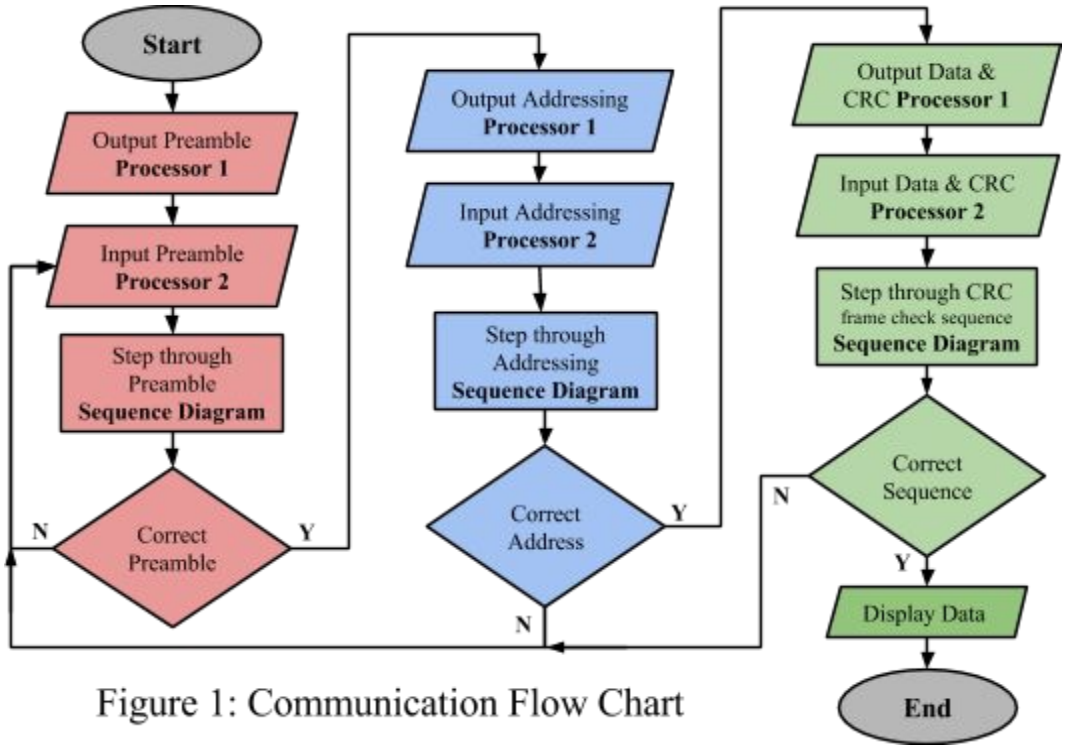

Figure 1: Communication Flow Chart

Theory:

The object of this system is accurately send and receive packets of data. Four main parts of any data communication are: Preamble, Addressing, Data, and CRC (Figure 2). Each of these segments are located in the data to be able to recognize when each segments will be seen at what time. These parts are essential to data communication systems, without all of these components receiving systems would have no idea what is data from the information they are receiving is.

Figure 2 helps to represent different slotted bits for each of the four main parts. In this figure Preamble, Addressing and Data are all represented by an 8-bit binary number, while the CRC function is only a 4-bit binary number. Further implementations can be to increase the number of data bits preceding each with 8-bits of data followed by 4-bits of CRC.

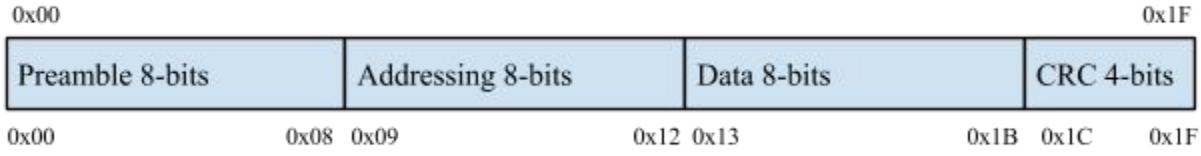

Figure 2: Transmission Layout

Interrupt:

Interrupts are an embedded programing system. This system is used when the current program needs to pause and do a sub method at possibly irregular iterations when a curtain value is changed. In the case of clock an interrupt is a helpful tool because any type a rising or falling edge is met particular methods want to be executed. [1]

Sequence Machine:

A sequencental machine helps to determine with the prediction of what the next state should be. This is an important step when creating a preamble function for a data communication system. Since a preamble function is typically a constant value and therefore will not change the state machine will determine which state is best appropriate for the following input value. Preamble is a very important step in any transmitting of data, it is used to help identify at what point is the beginning of all reliable information. In order to create a sequential machine the first step should be to create a state diagram (Figure 3).

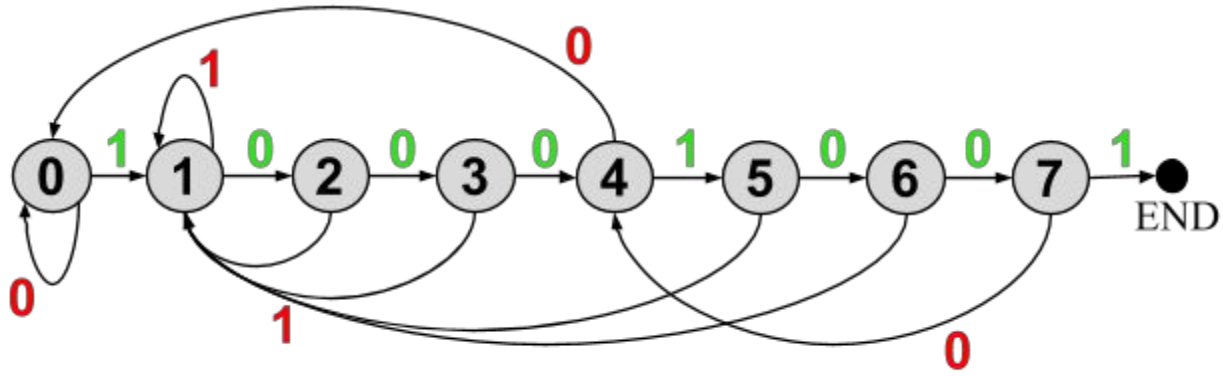

Figure 3: State Diagram for 10001001=0x89

Figure 3 demonstrates the state diagram for in sequence of 0x89 or in binary 10001001. Each state is represented by a circle with numbers ranging from 0 to 7, the arrows show at that state the direction to go given either a logic '1' or '0'. The red numbers represent the incorrect input sequence for the next state, the green numbers represent the correct sequence of steps. Once the state diagram is completed and verified, the next step is to create a table to better represent the next state process, which is represented on Table 1.

Table 1: State Table

Present State (Decimal)	Next State (Decimal)	
	Input = (0)	Input = (1)
0	0	1
1	2	1
2	3	1
3	4	1
4	0	5
5	6	1
6	7	1
7	4	END

Table 1 depicts present state and next state, with the next state depending upon what input happened at the current state. This sequence of steps will end only when it has found 0x89 within a inputted data group. Having a state table helps to better implement a sequence detector into any code form.

CRC:

Data gets transferred by all kinds of different computer processor. The important factor when data is being transmitted is the knowledge that the data has finished transmitting, this ability is the function of the CRC. The CRC process has two important parts for the transmitter and receiver processors. The transmitters CRC process begins once the data is initialized the CRC polynomial function is executed to calculate the frame check sequence (FCS). These bits are added on the end of the data so that for the receiver process will be able to step through the CRC polynomial and at the end of the frame check sequence process if the values aren't 0 then there was an error in the transmission of the data bits. An example for the CRC process can be seen in Equation 1, Figure 4 and Table 2.

$$Equation\ 1:\ CRC\ Polynomial = X^4 + X + 1$$

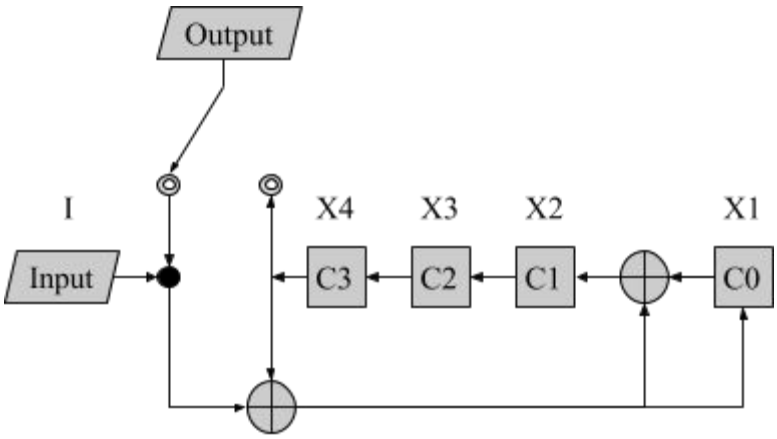

Figure 4: CRC Layout

Figure 4 depicts how the CRC Layout for the polynomial, use in Equation 1, is represented. The implementation of this from an data input to display the CRC process is seen in Table 2 with an input value of 0xAA. The frame check sequence (FCS) is represented in Table 2

as the highlighted value 0x1001. The end check is also seen on Table 1 with the end of the FCS giving each polynomial of the CRC the value 0 to be able to prove that the CRC was implemented correctly.

Table 2: CRC calculation table

C3←C2	C2←C1	C1	C0	C1=C0⊕(C3⊕I)	C0=C3⊕I	Input
0	0	0	0	1	1	1
0	0	1	1	1	0	0
0	1	1	0	1	1	1
1	1	1	1	0	1	0
1	1	0	1	1	0	1
1	0	1	0	1	1	0
0	1	1	1	0	1	1
1	1	0	1	0	1	0
1	**0**	**0**	**1**	1	0	**1**
0	0	1	0	0	0	**0**
0	1	0	0	0	0	**0**
1	0	0	0	0	0	**1**
0	**0**	**0**	**0**	**---**	**---**	**---**

Design:

Transmission:

In order to transmit the data we needed to create our own signal, high and low, that the could be easily changed with different data. To do this we decided to use the already available clock of the microprocessor to assist us in making the code work. Once the data is put in a specific variable, the system reads the variable and masks off the data bit by bit until it has all the bits of data separately in their own variable. Then after all the data has been processed we artificially create a clock by making and output high, then see if the first bit of data was high or low. If it was high we output a one otherwise we output zero. Then we put a delay to slow down our "clock", after the delay is over we set our artificial clock to low, then have another delay and move on to the next bit after it is over. The delay that is used can be adjusted and changed to the speed in which we want our system to move at. This continues until all the bits in preamble, addressing, data and CRC are processed.

This process of the artificial clock being passed through Preamble, Addressing, Data and CRC is representing in Figure 5, which is being observing through an oscilloscope.

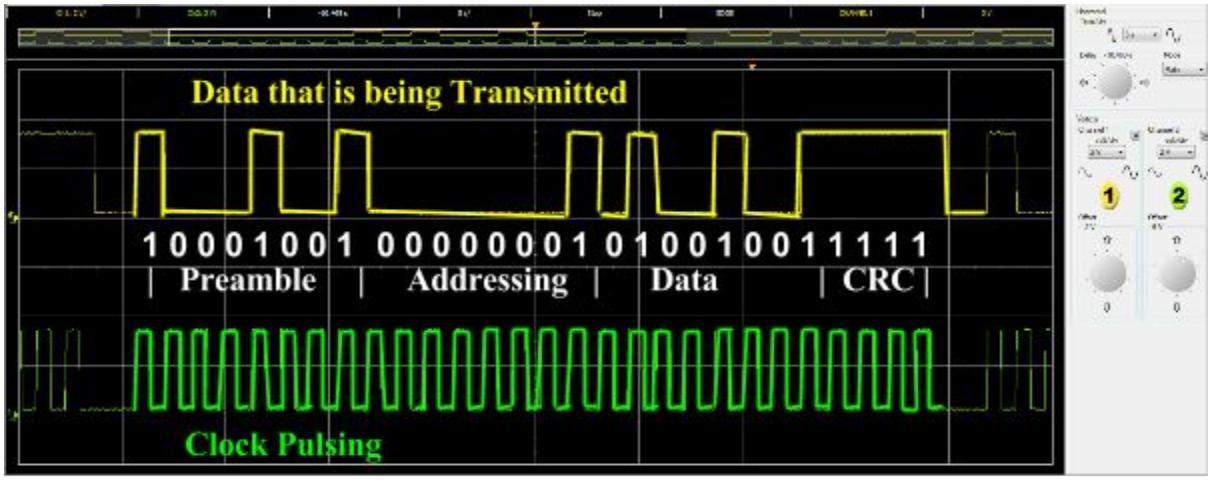

Figure 5: Transmission Groupings with Clock

The program for the main method of the transmission processor calls sub methods. one sub method is just used for the processor to display what data trial it is on. The other sub method has three parameters (Preamble, Addressing, Data). Each of these parameters can not be larger than an 8-bit value. There are also several different cases to transmit data in different forms by sending different addressing lines.

One main error that affects at least two lines of code would be if the preamble was incorrect and the data was correct preamble of 0x89. Since the program will always transmit all the three components if the same preamble was missed but was else where the code would relook for preamble and get the wrong value. This error is found on the receiving processor by never saving the addressing or data because something is out of sink and will return to waite for a correct preamble. To help eliminating losing 1 data segment can be implemented by transmitting back to processor 1 that something was wrong.

```
10  int main (void)
11  {
12    int num=0;
13
14    //Diplay what data point you are at
15    num=trial(num);
16
17    //Call method send with the parameters Preamble, Address, Data
18    send(0x89, 0xA1, 0xD5); //Bit Cast Addressing: Processor A1
19      num=trial(num);
20    send(0x89, 0xA2, 0xD5); //Bit Cast Addressing: Processor A2
21      num=trial(num);
22    send(0x89, 0xB1, 0xD5); //Bit Cast Addressing: Processor B3
23      num=trial(num);
24    send(0x89, 0xA7, 0xD5); //Bit Cast Addressing: Processor  A7
25      num=trial(num);
26    send(0x89, 0xAF, 0x89); //Multi Cast Addressing: Processor's A
27      num=trial(num);
28    send(0x89, 0xF0, 0xD5); //Bit Cast Addressing: Processor F0
29      num=trial(num);
30    send(0x89, 0xFF, 0xFF); //Broad Cast Addressing: All Processors
31      num=trial(num);
32  }
```

Receive:

To receive the transmission data first we made use of an interrupt look for the negative edge of our clock, this ensures that our transmission data is lined up correctly and our clocks between both processors, so that they are in sync. In the Freescale Arm book on page 205 there was an interrupt code that would be activated on the negative edge clock edge. This was the initial start ground of getting the receive code operational. The reasons for starting with this code was to be able to observe with the LED light on the processor that it was entering the interrupt on each time the clock went from high to low. So that it was easier to know when it would be reading the different information segments. [1]

The interrupt method starts on line 85, and toggles the internal **Red** LED light to help signify that the program is in the interrupt. To enter the interrupt the clock is sent to Port A[1] and if the clock goes from high to low then the program will immediately enter the interrupt. Once in the interrupt Port E[0] receives the data and this value is stored in an internal variable seq, as seen on lines 94 to 97.

```
85  void PORTA_IRQHandler(void) {
86    int i=100;
87      PTD->PDOR=0x1;//oscilloscope D0 represent time period
88    //We are recieving info
89    //Red light represents if in the interrupt or not
90        PTB->PDOR &= ~0x40000; /* turn on red 0x40000 LED */
91        delayMs(i);
92        PTB->PDOR |= 0x40000; /* turn off red 0x40000 LED */
93        delayMs(i);
94        if((PTE->PDIR&&0x1)==0x1)//Input to check if data is high or low
95          seq=1;
96        else
97          seq=0;
```

Once we start to receive the transmitted data we have to make sure we are receiving at the beginning of the packet (or preamble). This is done through the use of a state machine, the

state machine makes sure that we have the right data and starting at the right place. This state machine can be seen in Figure 3 and Table 1. The processor would not leave the preamble code until it was completed.

After we receive the preamble we then receive the address, if the address matches the address or group of the machine it will continue to receive data otherwise it will reset to start looking for a new packet. If the address is good, it will start to take in data which the system will hold and display on screen live.

The final part of the transmitted data is the actual data and CRC. As data is taken in the receiver is recalculating the CRC frame check sequence (FCS) that can be seen in Figure 4 and Table 2. If after the final FCS is calculated and all the polynomials are 0 data was correctly received and will be displayed.

Figure 6 helps to represent the clock that is being transmitted form Processor 1 **Yellow** and in Processor 2 will activate the interrupt. The other clock pulse is the entire time in the code that the processor is within the interrupt **Green**. This was to help check the timing element and check that we are not missing any transmitted clock cycles. Figure 6 also shows that the interrupt is active on the negative clock edge.

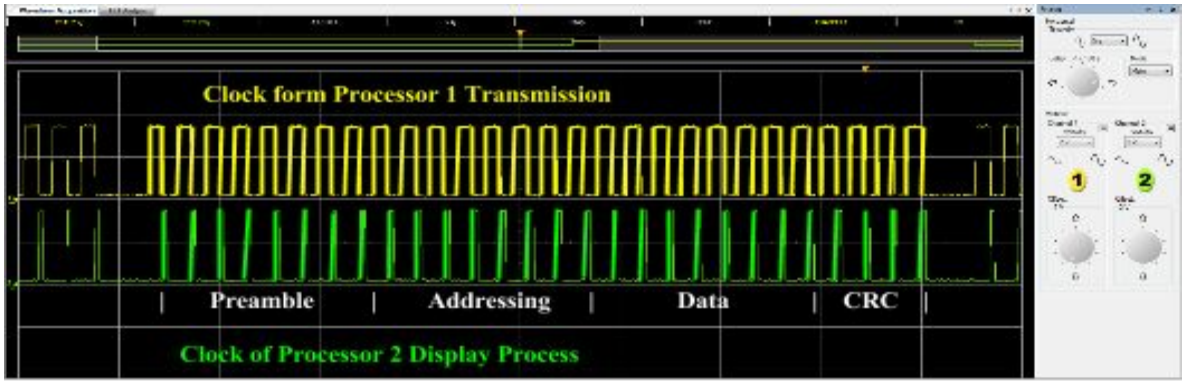

Figure 6: Clock of Both Processors

Preamble:

The preamble is done to have a known 8 bit number be checked by a sequence detector, so that it is known where the data actually starts. The sequence detector enables the program to check at what part the preamble is finished, this is represented in Figure 3 and Table 1.

Stages that were used in creating a preamble detector were to first check and receive the preamble. This is done through the use of an interrupt. The interrupt will give you the ability that at the time the clock is hit the display processor will jump to an interrupt and retrieve what the input is [1]. It will first go through the preamble sequence detector to find the completion of the preamble number (0x89). Then once the preamble is successful it will display both the preamble and data in a loop through the use of 8 led's.

The coded preamble process can be seen starting at line 104 in the code. In this process when the program is in preamble an external LED (which is red) will be on. This is done to help see at what stage of the program the code is located. After this lines 107 to 117 are representing a switch/case statement. This is depicting all the possible sequences needed to be applied in order to check that the right value is being passed through. This statement process is identical to the processes in Figure 2 and Table 1 the state diagram and table. When the present state is 7 all the 8 display LED's would turn on and the program would proceed to addressing.

```
103         //Preamble
104         if (pre==0)
105         {
106           PTB->PDOR=0x1;       //PortB[0] (Red LED) represents Preamble
107           switch(pstate)       //Sequence diagram for 0x89
108           {
109             case 0: {if(seq==0) nstate=0; else nstate=1; break;}
110             case 1: {if(seq==0) nstate=2; else nstate=1; break;}
111             case 2: {if(seq==0) nstate=3; else nstate=1; break;}
112             case 3: {if(seq==0) nstate=4; else nstate=1; break;}
113             case 4: {if(seq==0) nstate=0; else nstate=5; break;}
114             case 5: {if(seq==0) nstate=6; else nstate=1; break;}
115             case 6: {if(seq==0) nstate=7; else nstate=1; break;}
116             case 7: {if(seq==0) nstate=4; else nstate=1; break;}
117           }
118           PTC->PDOR=nstate;
119
120           if(pstate==7) //At the end of the sequence diagram display 0xFF and reset LEDs
121           {
122             PTC->PDOR=0xFF;
123             delayMs(100);
124             PTC->PDOR=0x0;
125
126             pre=1;           // We are done with preamble
127           }
128           pstate=nstate;     //Save next state for present state
129         }
```

The preamble involved creating a signal to be sent with the clock. The clock is represented through each iteration that the clock makes, this would be executed by one line of code from the program. This was constructed by every line of code is sending the value and the clock pulse then the clock pulse is inverted but the value stays the same until the next line when new data is entered.

Addressing:

Addressing is a process that is done after preamble but before the data is sent. This would check and see where the data would be transmitted to. The process for writing this is similar to receiving the data by counting the number of positions that have been passed and if this lines up with a correct address or group then the process would be correct and the system would continue on, otherwise the system will reset back to looking for a new packet. The addressing is important so that communication can happen between different and multiple boards.

The code segment below starting at 132 is the addressing part of the program. When the program is in addressing an external LED (which is blue) will be on. This is done to help see at what stage of the program the code is located. Then the address gets stored through a shift function to retrieve all 8 bits. Once it has completed this process 8 times the address will be checked with the available addresses to this program. If none of these match the current information will be scrapped and the program will start back in the preamble phase.

```
132         //Addressing
133         if(pr==1)
134         {
135           PTB->PDOR=0x2;     //PortB[1] (Blue LED) represents Addressing
136           data=data*2+seq;   //Storing address through a shift
137           PTC->PDOR=data;
138           delayMs(100);
139           pos++;
140           //0xA1---Bit Cast Addressing
141           //0xFF---Broad Cast Addressing
142           //0xAF---Multi Cast Addressing
143           if((pos==8)&&((data==0xA1)||(data==0xFF)||((data&0xFF)==0xAF)))
144           {
145             dc=1;
146             PTC->PDOR=0x0;
147             pr=0;
148             p=1;
149             data=0;
150             pos=0;
151           }
152           if(pos>8) //Didn't retrieve address start over
153           {
154             PTD->PDOR=0x0;
155             //ZEROED OUT
```

The process of addressing data to different processors involves changing what the address the processor is. In the case of three processors the if statements are controlled by the first 4-bits of the address will control which processor group the data will be transmitted to. The second 4-bits of the address are used to help determine in which group of processors will get the data. These help to have bit and multi casting abilities for the data communication. If in the case the person wanted to transmit all the processors then the address would be 0xFF. Examples of this coding scheme for different processors can be seen below.

```
//0xA1---Bit Cast Addressing
//0xFF---Broad Cast Addressing
//0xAF---Multi Cast Addressing
if((pos==8)&&((data==0xA1)||(data==0xFF)||((data&0xFF)==0xAF)))

//0xA2---Bit Cast Addressing
//0xFF---Broad Cast Addressing
//0xAF---Multi Cast Addressing
if((pos==8)&&((data==0xA2)||(data==0xFF)||((data&0xFF)==0xAF)))

//0xB1---Bit Cast Addressing
//0xFF---Broad Cast Addressing
//0xBF---Multi Cast Addressing
if((pos==8)&&((data==0xB1)||(data==0xFF)||((data&0xFF)==0xBF)))
```

CRC:

CRC is used to be able to check that the entire data has been sent through correctly. In our process we were using a polynomial from Equation 1 and Figure 4 to represent the logic layout for this polynomial. This Equation would calculate on the transmitter and send the values at the end. On the receiver side the device would get the data, recalculate the CRC and check it with the CRC 4-bits that came through. If these were identical the final value would be 0 and the data would have been transmitted correctly. A sample of our CRC is shown in Table 2 for the input value of 0xAA or in binary 10101010.

In the code the data and CRC segment starts on line 173. Once the program is in data and CRC an external LED (which is green) will be on. This is done to help see at what stage of the program the code is located. Once inside data the similar process of extracting the data is the same as in addressing by using a shifter and then displayed. In data it is also important to check and see when the entire data has been transmitted. The CRC is therefore making checks as each bit of data is transmitted into it. The steps that is goes through is the same steps as in Figure 4 and Table 2 for the polynomial in Equation 1 ($X^4 + X + 1$).

```
173         //Data and CRC
174         if(dcr==1)
175         {
176           PTB->PDOR=0x4;    //PortB[2] (Green LED) represents Data & CRC
177           data=data*2+seq;  //Storing data through a shift
178           PTC->PDOR=data;   //Display data values
179
180           //CRC check
181           Y=C3 ^ seq;
182           C3=C2;
183           C2=C1;
184           C1=Y ^ C0;
185           C0=Y;
186
187           //Pos 11 is the end of data and one before end of CRC
188           if((pos==11)&&(C3==0)&&(C2==0)&&(C1==0)&&(C0==0))
189           {
190             PTB->PDOR=0x7;  //Red, Blue, and Green LED's on represent Good Data received
191             PTC->PDOR=0x0;
192             delayMs(50);
193             data=data/16;   //Retreive actual data
194             PTC->PDOR=data; //Display 8-bit data
195             delayMs(300);
196
197             //ZEROED OUT
```

When all the bits have been transmitted the position and the CRC is checked if all the polynomials values are 0, then the data will be displayed again and all the external LED will turn on to signal that the process was completed successfully. It will then reset all the values to wait for another transmission. If however position was too large and the CRC didn't check then the processor would just reset everything because there was some kind of error.

Problems:

Timing:

Timing held several problems, when we were trying at one point to be able to set up the internal clock, the timing of this was seeming to be slower than our processes. To get this to work we were thinking of putting a delay in but this could not have both our processors clocks lineup and with that we changed the set up. We decided to use an interrupt to trigger the received data sent by the receiver. The trigger is a clock that we decided to make using the ARM boards main clock. Since the processors were all ARM boards there timers would all be the same we would just add the same delay to help slow them down.

```
226             //ZEROED OUT
227             data=0;
228             pstate=0;
229             nstate=0;
230             pos=0;      //Position is set back to zero
231             pr=0;
232             pre=0;
233             p=0;
234             dc=0;
235             dcr=0;
236
237             seq=0;    //Need to reset the seq.
238
239             C0=0;
240             C1=0;
241             C2=0;
242             C3=0;
243             Y=0;
244           }
245         }
246       }
247
248       if((p==0)&&(pre==1))// Doing addressing
249         {
250           pr=1;
251         }
252       if(dc==1)    //Doing data
253         {
254           dcr=1;
255         }
```

The problem with using the interrupt then was that when the interrupt was activated it would leave the main code and then return back to the main code where it have previously left. At one point in our main code we had placed lines 248 to 255. The problem that would be sometimes occurring was that we would complete data and zero everything out, as seen in lines 226 through 227, but if the interrupt had left the main program at line 253 ‘dcr’ would still be 1. This value would have us continue to perform data calculations when we should be back in preamble. This error helped us to reunderstand what and how an interrupt actually operates.

The zeroing out function also is present after addressing if the addressing was incorrect although for this zeroed out only line 237 was not inputted incase the current sequence that it had inputted was infact preamble.

Preamble:

Several times because the delay time was so long there data and preamble were mixed up. The receive system had problems getting out of the preamble state machine. Once we re-adjusted our our delays, not only made them smaller but made the delays the same on both the send and receive systems. Before we made our delays the same most of our problems came from the fact that the transmission system delay was in cycles and the receive system was in milliseconds. After this situation was corrected the preamble worked perfectly.

These two different delays sub methods are from the transmission and receive code. Another importance involving the receive delay was to make sure that in each section the total delay time that you would be inside the interrupt would not exceed the delay for one clock cycle. It is important because if the interrupt delay was longer you could miss several transmitted data. This is another reason to check the clock pulse throughout the entire interrupt as seen in Figure 6.

Transmission Main Delay

```
150  void delay(void)     //0.7 Second delay for each bit
151  {
152    int n=700;
153    int i;
154    int j;
155    n=n/2;
156    for(i = 0 ; i < n; i++)
157    for (j = 0; j < 7000; j++) {}
158    }
```

Receive Delay

```
263  void delayMs(int n) {
264
265    int i;
266    int j;
267    n=n/2;
268    for(i = 0 ; i < n; i++)
269    for (j = 0; j < 7000; j++) {}
270
271  }
```

Conclusion:

Our project was to create a packet sender and receiver unit that would be able to send a Preamble, Addressing, Data and CRC across the line to the different receivers. The transmitter sends the data perfectly no matter what data is sent to it as long as it is the right size. The receiver will read in the data, bit by bit as it is sent syncing the clocks by using an interrupt. Our system cannot only detect and process data from packets, it also can check the data using CRC to verify that the data is correct. If there is some miscommunications then it does not save the data. The system also checks the address of the data being sent to make sure that data is for it. With the addressing we can assign each device an individual address and group, so that the the data could be accepted if it is for its address, group, or for everyone.

Our system is very advanced if it does not pass the addressing check or CRC the data is not processed and is zeroed out. At this point the system would start over. Our system works exceptionally well, the only thing that we decided not to include was error correction, due to not having enough time, otherwise it would have been implemented. Both our microcontrollers operate just as they should and our code works amazingly well. Figure 7 helps to show the final processor system with 4 processors 1 is transmitting information to the other 3 processors of different addresses. Each of these processors are able to save the data that they were supposed to receive at that time.

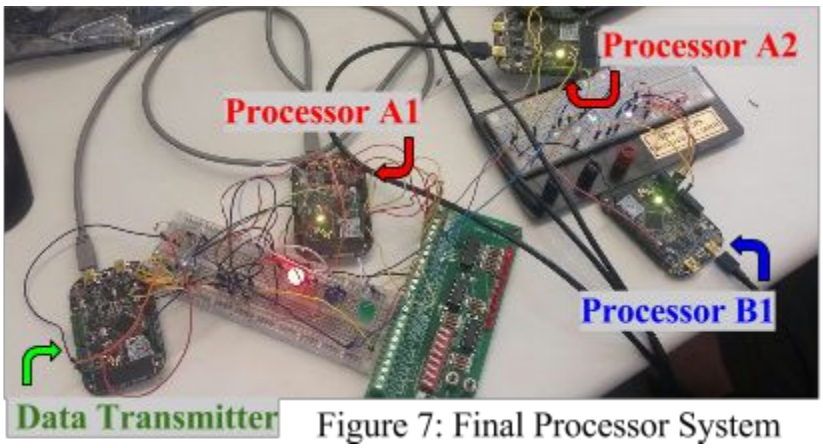

Figure 7: Final Processor System

Works Cited:

[1] Muhammad Ali Mazidi, *Freescale ARM Cortex-M Embedded Programming,*

Seattle, WA: Amazon, 2014

Appendix:

Transmission Code:

```
#include <MKL25Z4.H>

void delay(void);
void Delay(void);                   //Delay on for end of each data segment
void crc(int DATA);
int cr[4];
void send(int PRE, int Address, int DATA);

int trial(int A);
int main (void)
{
    int num=0;

    //Display what data point you are at
    num=trial(num);

    //Call method send with the parameters Preamble, Address, Data
    send(0x89, 0xA1, 0xD5);  //Bit Cast Addressing: Processor A1
        num=trial(num);
    send(0x89, 0xA2, 0xD5);  //Bit Cast Addressing: Processor A2
        num=trial(num);
    send(0x89, 0xB1, 0xD5);  //Bit Cast Addressing: Processor B3
        num=trial(num);
    send(0x89, 0xA7, 0xD5);  //Bit Cast Addressing: Processor  A7
        num=trial(num);
    send(0x89, 0xAF, 0x89);         //Multi Cast Addressing: Processor's A
        num=trial(num);
    send(0x89, 0xF0, 0xD5);         //Bit Cast Addressing: Processor F0
        num=trial(num);
    send(0x89, 0xFF, 0xFF);         //Broad Cast Addressing: All Processors
        num=trial(num);
}
//Trial gets inputed a number, Adds 1 and displays the number in PortD
//This is to show what number of data you are transmitting
int trial(int A)
{
    SIM->SCGC5=0x3D80;
    PORTC->PCR[0]=0x100;//Display on LED lights
    PORTC->PCR[1]=0x100;
    PORTC->PCR[2]=0x100;
    PORTC->PCR[3]=0x100;
    PTC->PDDR=0xFFFFFFFF;

    A++;
    PTC->PDOR=A;
    return A;
}

//Input the Preamble Address and Data
void send(int PRE, int Address, int DATA)
{
```

```
//Preamble inputs
    int a=0;
    int b=0;
    int c=0;
    int d=0;
    int e=0;
    int f=0;
    int g=0;
    int h=0;
//Data Inputs
    int i=0;
    int j=0;
    int k=0;
    int l=0;
    int m=0;
    int n=0;
    int o=0;
    int p=0;
//Address inputs
    int A=0;
    int B=0;
    int C=0;
    int D=0;
    int E=0;
    int F=0;
    int G=0;
    int H=0;
//Set Preamble
    if ((PRE&0x80)==0x80) a=0x1;
    if ((PRE&0x40)==0x40) b=0x1;
    if ((PRE&0x20)==0x20) c=0x1;
    if ((PRE&0x10)==0x10) d=0x1;
    if ((PRE&0x08)==0x8) e=0x1;
    if ((PRE&0x04)==0x4) f=0x1;
    if ((PRE&0x02)==0x2) g=0x1;
    if ((PRE&0x01)==0x1) h=0x1;
//Set Addressing
    if ((Address&0x80)==0x80) A=0x1;
    if ((Address&0x40)==0x40)B=0x1;
    if ((Address&0x20)==0x20)C=0x1;
    if ((Address&0x10)==0x10) D=0x1;
    if ((Address&0x08)==0x8) E=0x1;
    if ((Address&0x04)==0x4) F=0x1;
    if ((Address&0x02)==0x2) G=0x1;
    if ((Address&0x01)==0x1) H=0x1;

//Calculate the CRC
    crc(DATA);

//Set Data
    if ((DATA&0x80)==0x80) i=0x1;
    if ((DATA&0x40)==0x40) j=0x1;
    if ((DATA&0x20)==0x20) k=0x1;
    if ((DATA&0x10)==0x10) l=0x1;
    if ((DATA&0x08)==0x8) m=0x1;
    if ((DATA&0x04)==0x4) n=0x1;
    if ((DATA&0x02)==0x2) o=0x1;
    if ((DATA&0x01)==0x1) p=0x1;
```

```
SIM->SCGC5=0x3D80;

PORTE->PCR[0]=0x100;    //DATA Port
PORTE->PCR[1]=0x100;    //CLK Port
PTE->PDDR=0xFFFFFFFF;
PTE->PDOR=0x00;

//Preamble Clock
    PTE->PDOR=0x2|(0x1&a); delay();
    PTE->PDOR=0x0|(0x1&a); delay();
    PTE->PDOR=0x2|(0x1&b); delay();
    PTE->PDOR=0x0|(0x1&b); delay();
    PTE->PDOR=0x2|(0x1&c);delay();
    PTE->PDOR=0x0|(0x1&c); delay();
    PTE->PDOR=0x2|(0x1&d);delay();
    PTE->PDOR=0x0|(0x1&d); delay();
    PTE->PDOR=0x2|(0x1&e);delay();
    PTE->PDOR=0x0|(0x1&e); delay();
    PTE->PDOR=0x2|(0x1&f);delay();
    PTE->PDOR=0x0|(0x1&f); delay();
    PTE->PDOR=0x2|(0x1&g);delay();
    PTE->PDOR=0x0|(0x1&g); delay();
    PTE->PDOR=0x2|(0x1&h);delay();
    PTE->PDOR=0x0|(0x1&h); delay();

//Address Clock
    PTE->PDOR=0x2|(0x1&A); delay();
    PTE->PDOR=0x0|(0x1&A); delay();
    PTE->PDOR=0x2|(0x1&B); delay();
    PTE->PDOR=0x0|(0x1&B); delay();
    PTE->PDOR=0x2|(0x1&C);delay();
    PTE->PDOR=0x0|(0x1&C); delay();
    PTE->PDOR=0x2|(0x1&D);delay();
    PTE->PDOR=0x0|(0x1&D); delay();
    PTE->PDOR=0x2|(0x1&E);delay();
    PTE->PDOR=0x0|(0x1&E); delay();
    PTE->PDOR=0x2|(0x1&F);delay();
    PTE->PDOR=0x0|(0x1&F); delay();
    PTE->PDOR=0x2|(0x1&G);delay();
    PTE->PDOR=0x0|(0x1&G); delay();
    PTE->PDOR=0x2|(0x1&H);delay();
    PTE->PDOR=0x0|(0x1&H); delay();

//Data Clock
    PTE->PDOR=0x2|(0x1&i);delay();
    PTE->PDOR=0x0|(0x1&i); delay();
    PTE->PDOR=0x2|(0x1&j);delay();
    PTE->PDOR=0x0|(0x1&j); delay();
    PTE->PDOR=0x2|(0x1&k);delay();
    PTE->PDOR=0x0|(0x1&k); delay();
    PTE->PDOR=0x2|(0x1&l);delay();
    PTE->PDOR=0x0|(0x1&l); delay();
    PTE->PDOR=0x2|(0x1&m);delay();
    PTE->PDOR=0x0|(0x1&m); delay();
    PTE->PDOR=0x2|(0x1&n);delay();
    PTE->PDOR=0x0|(0x1&n); delay();
    PTE->PDOR=0x2|(0x1&o);delay();
    PTE->PDOR=0x0|(0x1&o); delay();
    PTE->PDOR=0x2|(0x1&p);delay();
    PTE->PDOR=0x0|(0x1&p); delay();
```

```
    //CRC clock
        PTE->PDOR=0x2|(0x1&cr[3]);delay();
        PTE->PDOR=0x0|(0x1&cr[3]); delay();
        PTE->PDOR=0x2|(0x1&cr[2]);delay();
        PTE->PDOR=0x0|(0x1&cr[2]); delay();
        PTE->PDOR=0x2|(0x1&cr[1]);delay();
        PTE->PDOR=0x0|(0x1&cr[1]); delay();
        PTE->PDOR=0x2|(0x1&cr[0]);delay();
        PTE->PDOR=0x0|(0x1&cr[0]); delay();

        PTE->PDOR=0x0;
        Delay();
}

void delay(void)       //0.7 Second delay for each bit
{
    int n=700;
    int i;
    int j;
    n=n/5;
    for(i = 0 ; i < n; i++)
    for (j = 0; j < 7000; j++) {}
}

    void Delay(void)    //2 Second delay for end of process
{
    int n=2000;
    int i;
    int j;
    n=n/5;
    for(i = 0 ; i < n; i++)
    for (j = 0; j < 7000; j++) {}
}
void crc(int DATA)     //Calculating CRC =X^4+X^1+1
{
    int C0=0;
    int C1=0;
    int C2=0;
    int C3=0;
    int Y=0;
    int I;
    int Z[8];

    int i=0;
    int j=0;
    int k=0;
    int l=0;
    int m=0;
    int n=0;
    int o=0;
    int p=0;

    //Set data for CRC
    if ((DATA&0x80)==0x80) i=0x1;
    if ((DATA&0x40)==0x40) j=0x1;
    if ((DATA&0x20)==0x20) k=0x1;
    if ((DATA&0x10)==0x10) l=0x1;
    if ((DATA&0x08)==0x8) m=0x1;
    if ((DATA&0x04)==0x4) n=0x1;
    if ((DATA&0x02)==0x2) o=0x1;
    if ((DATA&0x01)==0x1) p=0x1;
```

```
    //Saving the inputs, data to an array
    Z[0]=i;
    Z[1]=j;
    Z[2]=k;
    Z[3]=l;
    Z[4]=m;
    Z[5]=n;
    Z[6]=o;
    Z[7]=p;

    //Calculate CRC through XOR
    for(I=0; I<8;I++)
    {

        Y=C3 ^ Z[I];
        C3=C2;
        C2=C1;
        C1=Y ^ C0;
        C0=Y;
    }
    //Save CRC to Global variables
    cr[0]=C0;
    cr[1]=C1;
    cr[2]=C2;
    cr[3]=C3;
}
```

Receive Code:

```
#include "MKL25Z4.h"
void delayMs(int n);
//Global Variables
int data=0;
int seq;
//Preamble variables
int pstate=0;
int nstate=0;
//Position
int pos=0;
int pr=0;
int pre=0;
int p=0;
int dc=0;
int dcr=0;

//Used for CRC checking
int C0=0;
int C1=0;
int C2=0;
int C3=0;
int Y=0;
int I;

int main(void) {
    __disable_irq(); /* disable all IRQs */

    SIM->SCGC5=0x3D80;

    SIM->SCGC5 |= 0x400; /* enable clock to Port B */
    PORTB->PCR[18] = 0x100; /* make PTB18 pin as GPIO */
    PORTB->PCR[19] = 0x100; /* make PTB19 pin as GPIO */
    PTB->PDDR |= 0xC0000; /* make PTB18, 19 as output pin */
    PTB->PDOR |= 0xC0000; /* turn off LEDs */
    SIM->SCGC5 |= 0x200; /* enable clock to Port A */
    /* configure PTA1 for interrupt */
    PORTA->PCR[1] |= 0x00100; /* make it GPIO */
    //PORTA->PCR[1] |= 0x00003; /* enable pull-up */
    PTA->PDDR &= ~0x0002; /* make pin input */
    PORTA->PCR[1] &= ~0xF0000; /* clear interrupt selection */
    PORTA->PCR[1] |= 0xA0000; /* enable falling edge interrupt */
    // configure PTA2 for interrupt
    PORTA->PCR[2] |= 0x00100; /* make it GPIO */
    PORTA->PCR[2] |= 0x00003; /* enable pull-up */
    PTA->PDDR &= ~0x0004; /* make pin input */
    PORTA->PCR[2] &= ~0xF0000; /* clear interrupt selection */
    PORTA->PCR[2] |= 0xA0000; /* enable falling edge interrupt */
    NVIC->ISER[0] |= 0x40000000; /* enable INT30 (bit 30 of ISER[0]) */
    __enable_irq(); /* global enable IRQs */


    PORTE->PCR[0]=0x100;//DATA
    PTE->PDDR=0xFFFFFF00;

    //Turning on the LED's
    PORTC->PCR[0]=0x100;//Display on LED board
    PORTC->PCR[1]=0x100;
    PORTC->PCR[2]=0x100;
    PORTC->PCR[3]=0x100;
```

```
    PORTC->PCR[4]=0x100;//Display on LED board
    PORTC->PCR[5]=0x100;
    PORTC->PCR[6]=0x100;
    PORTC->PCR[7]=0x100;
    PTC->PDDR=0xFFFFFFFF;

    PORTB->PCR[0]=0x100;//3 different LED
    PORTB->PCR[1]=0x100;
    PORTB->PCR[2]=0x100;
    PTB->PDDR=0xFFFFFFFF;

    PORTD->PCR[0]=0x100;//Port D is used for the oscilloscope
    PTD->PDDR=0xFFFFFFFF;

    /* toggle the red LED continuously */
    while(1) {
    //    PTB->PTOR |= 0x40000; /* toggle red LED */
        //delayMs(100);


        ;
    }
}

/* A pushbutton switch is connecting either PTA1 or PTA2 to ground to trigger PORTA interrupt
*/
void PORTA_IRQHandler(void) {
    int i=100;
        PTD->PDOR=0x1;//oscilloscope D0 represent time period
    //We are receiving info
    //Red light represents if in the interrupt or not
                PTB->PDOR &= ~0x40000; /* turn on red 0x40000 LED */
                delayMs(i);
                PTB->PDOR |= 0x40000; /* turn off red 0x40000 LED */
                delayMs(i);
                if((PTE->PDIR&&0x1)==0x1)//Input to check if data is high or low
                {
                        seq=1;
                }
                else
                {
                        seq=0;
                }

                //Preamble
                if (pre==0)
                {
                        PTB->PDOR=0x1;                //PortB[0] (Red LED) represents Preamble
                        switch(pstate)                //Sequence diagram for 0x89
                        {
                                case 0: {if(seq==0) nstate=0; else nstate=1; break;}
                                case 1: {if(seq==0) nstate=2; else nstate=1; break;}
                                case 2: {if(seq==0) nstate=3; else nstate=1; break;}
                                case 3: {if(seq==0) nstate=4; else nstate=1; break;}
                                case 4: {if(seq==0) nstate=0; else nstate=5; break;}
                                case 5: {if(seq==0) nstate=6; else nstate=1; break;}
                                case 6: {if(seq==0) nstate=7; else nstate=1; break;}
                                case 7: {if(seq==0) nstate=4; else nstate=1; break;}
                        }
                        PTC->PDOR=nstate;
```

```
        if(pstate==7) /End of the sequence display 0xFF and reset LEDs
        {
                PTC->PDOR=0xFF;
                delayMs(100);
                PTC->PDOR=0x0;

                pre=1;        // We are done with preamble
        }
        pstate=nstate;         //Save next state for present state
}

//Addressing
if(pr==1)
{
        PTB->PDOR=0x2;         //PortB[1] (Blue LED) represents Addressing
        data=data*2+seq;      //Storing address through a shift
        PTC->PDOR=data;
        delayMs(100);
        pos++;
        //0xA1---Bit Cast Addressing
        //0xFF---Broad Cast Addressing
        //0xAF---Multi Cast Addressing
        if((pos==8)&&((data==0xA1)||(data==0xFF)||((data&0xFF)==0xAF)))
        {
                dc=1;
                PTC->PDOR=0x0;
                pr=0;
                p=1;
                data=0;
                pos=0;
        }
        if(pos>8)    //Didn't receive address start over
        {
                PTD->PDOR=0x0;
                //ZEROED OUT
                data=0;
                pstate=0;
                nstate=0;
                pos=0;
                pr=0;
                pre=0;
                p=0;
                dc=0;
                dcr=0;

                C0=0;
                C1=0;
                C2=0;
                C3=0;
                Y=0;
        }
}

//Data and CRC
if(dcr==1)
{
        PTB->PDOR=0x4;         //PortB[2] (Green LED) represents Data & CRC
        data=data*2+seq;     //Storing data through a shift
        PTC->PDOR=data;        //Display data values
```

```
//CRC check
Y=C3 ^ seq;
C3=C2;
C2=C1;
C1=Y ^ C0;
C0=Y;

//Pos 11 is the end of data and one before end of CRC
if((pos==11)&&(C3==0)&&(C2==0)&&(C1==0)&&(C0==0))
{
        PTB->PDOR=0x7; //Red, Blue, and Green LED's on Good Data
        PTC->PDOR=0x0;
        delayMs(50);
        data=data/16;         //Receive actual data
        PTC->PDOR=data; //Display 8-bit data
        delayMs(300);

        //ZEROED OUT
        data=0;
        pstate=0;
        nstate=0;
        pos=0;                //Position is set back to zero
        pr=0;
        pre=0;
        p=0;
        dc=0;
        dcr=0;

        seq=0;         //Need to reset the seq.

        C0=0;
        C1=0;
        C2=0;
        C3=0;
        Y=0;
        delayMs(100);
        PTC->PDOR=0x0;
}
else                  //If data hasn't finished
{
        pos=pos+1;
//If the position is too large and CRC didn't check start over
        if((pos>=12)&&((C3!=0)||(C2!=0)||(C1!=0)||(C0!=0)))
        {
                PTD->PDOR=0x0;
                PTC->PDOR=0x0;
                //ZEROED OUT
                data=0;
                pstate=0;
                nstate=0;
                pos=0;                //Position is set back to zero
                pr=0;
                pre=0;
                p=0;
                dc=0;
                dcr=0;

                seq=0;         //Need to reset the seq.
```

```
                                        C0=0;
                                        C1=0;
                                        C2=0;
                                        C3=0;
                                        Y=0;
                                }
                        }
                }

                if((p==0)&&(pre==1))// Doing addressing
                        {
                                pr=1;
                        }
                if(dc==1)       //Doing data
                        {
                                dcr=1;
                        }

    PORTA->ISFR = 0x00000006; /* clear interrupt flag */
}

// Delay n milliseconds
// The CPU core clock is set to MCGFLLCLK at 41.94 MHz in SystemInit().
void delayMs(int n) {

    int i;
    int j;
    n=n/5;
    for(i = 0 ; i < n; i++)
    for (j = 0; j < 7000; j++) {}

}
```