Report for Coursework ELEC3227 Embedded Network System

Name: Travis Lam Han Yuen
Student ID: 30582105
Team C2

1. Introduction

This coursework aims to design the network architecture of a Philip hue lightning system to allow user to change the lightning of different patterns. It is implemented on 3 Il Matto board with RFM12B-S2 radio module. When a button on Il Matto board is pressed, a message should send to the destinated ilmatto board and the LED of the Il Matto lights up. The protocol layer is divided to 5 layers and each teammate responsible for one or two layer shown in figure 1. I work on Data Link and Physical Layer.

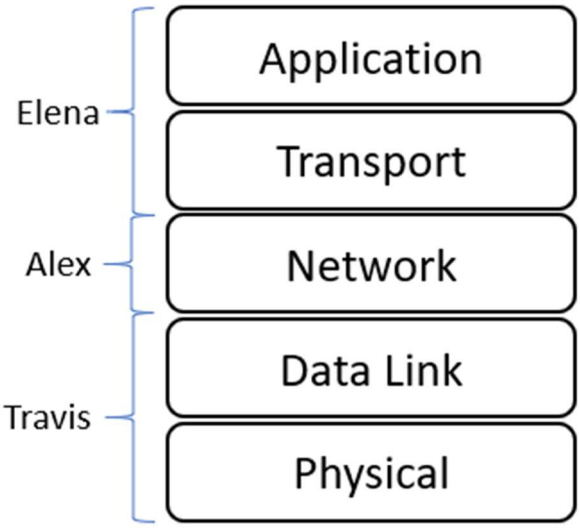

Figure 1. Protocol Layer and work assignmet

2. Standard Documentation

This document standardises Data Link Layer and Physical Layer of an embedded network architecture. An acknowledged connectionless service is used whereby receiver sends acknowledgement to transmitter when it receives data correctly.

2.1. Standardisation on Data Link Layer

DLL:	Header [1]	Control [2]	Addressing [2]	Length [1]	NET Packet (or part of) [1-23]	Checksum [2]	Footer [1]

Figure 2 Frame field structure of Data Link Layer

Figure 1 shows frame field structure of a Data Link Layer. In the data link layer, a frame is encapsulated by its header and footer, within the frame it contains control, addressing, length,

NETwork packet and checksum error. The number below each fragment represent the number of bytes it contains.

2.1.1. Header/Footer/Flag Byte

A framing method of byte stuffing will be used in this system. The header and footer will be used as flag byte to frame the bitstream. By using flag byte, Data Link Layer recognise specific bit pattern as a header and the bit pattern appears again at the end of the frame as a footer. The header and footer byte used in this system is 0x7E with the escape byte is 0x55. On transmitting, 0x55 will be added every 0x7E as stuff to escape flag byte appear at the middle of frame. On receiving, 0x55 will be removed to recover the data.

2.1.2. Control Byte (2 Bytes)

Sequence Number [0-3]	Acknowledge bits [4-7]	Checksum Method [8-11]	Number of Frame leave for receiving [12-15]

Table 1 Control Bits index allocation

Table 1 shows how the control byte is being used in the system, the 4-bits simple handshake protocol is implemented as a method of flow control in this system. The first 4 bits in the first byte is for number sequencing, so there will be 16 transmission before signal reset, while second 4 bits are used as acknowledge bit. Bit index at 8-11 is for switching the checksum method for example 00 is for fletcher checksum and 11 is for CRC checksum. Bit index 12-15 used to indicate the number of the packet left for receiving, this prevent receiver stops receiving when the NET packet is not completing its transmission.

2.1.3. Addressing (2 Bytes)

Source Address [4-7]	Destination Address [8-11]

Table 2 Addressing Bits allocation

By extracting addressing data from network layer, the first byte of the addressing byte is used to store the source address, whereas second byte is used to store the destination address, the address for each node will be node 1:0x00, node 2:0x05, node 3:0x0E.

2.1.4. Length (1 Byte)

The length byte simply represents the length of NET packet passed from upper layer.

2.1.5. NET Packet (1-23 Bytes)

The NET packet contains the piece of fragment in NET frame passed from Network layer, maximum size of NET packet is 23 Bytes. If the packet received from network layer is over 23 Bytes, it is then breakdown to separate piece and reassemble at the receiver.

2.1.6. Checksum (2 Bytes)

The checksum contains the bits used for error detection in NET packet. Primarily, fletcher checksum method will be used. The first byte will be sum of NET packet and Mod256 to give the first byte, while the second byte takes the cumulative sum of the NET packet and Mod256. The checksum packet aims to check if a packet transmits correctly, if error occurs it will resend the message again.

2.1.7. MAC sublayer

0-persistent will be used to broadcast the data to each node so that each broadcast contains its unique address for a node to listen to. When the node is ready to transmit it sense the channel whether it is idle, if it is idle then transmits immediately. Otherwise, it holds for a random period before transmission again.

2.1.8. Communication Between Layers

The service primitives offer by the DLL layer is from_network_layer(*frame) and a to_physical_layer(*frame) function. Function from_network_layer(*frame) is called to receive the packet from network layer and start to encode the Network packet. If a frame needed to pass to the lower layer, to_physical_layer(*frame) need to be call by the upper layer, it will return the packet to the lower layer.

2.2. Standardisation on Physical Layer

The PHY mainly uses function from RFM12B-S2 library. In this system, three Ilmatto boards is used as nodes to communicate with each board. The system broadcasts between each node using radio frequency operates at frequency band 868MHz.

On the board port B will be connect to the radio module RFM12B-S2, while the LED and the button will be connected to Pin num 7 for LED and Pin num 6 for Switch for each node.

3. Design

To make the network architecture of different layers. Class feature of C++ is used for each layer, with each layer features stored as members and member functions.

3.1. Typedef Packet data stream

To store the byte stream data of the packet of the layer, a typedef struct is used to define the packet. The members inside the struct packet is the component byte field component of data link layer stored as uint8_t array. A final uint8_t array with variable name: Everything is used to store every byte of the DLL component including Header, Control, Address, Length, NET packet, Checksum, Footer.

3.2. Data Link Layer

Class is used to code Data Link Layer, with correspondent members and member functions inside the class. The control components such as Checksum Method, Acknowledgement and is stored as a class member.

3.2.1. NET packet breakdown

When the data packet is passed from Network layer, the size of Network Packet is 128 bytes which exceeds the size of the packet allowed to pack in one frame, therefore it needs to be break down to 23 bytes each. The number of frames that need to be sent inside control bytes[12-15] is calculated based on this. When the NET packet received from Network Layer, it is stored inside a NET_queue uint8_t array, a class member of uint_8t read_until_index is used to keep track of the index where NET-queue is read and stored in DLL packet. On receiving, similar process to assemble the NET_queue data before it is passed back to Network Layer. The flow of NET packet breakdown can be shown in Figure 2.

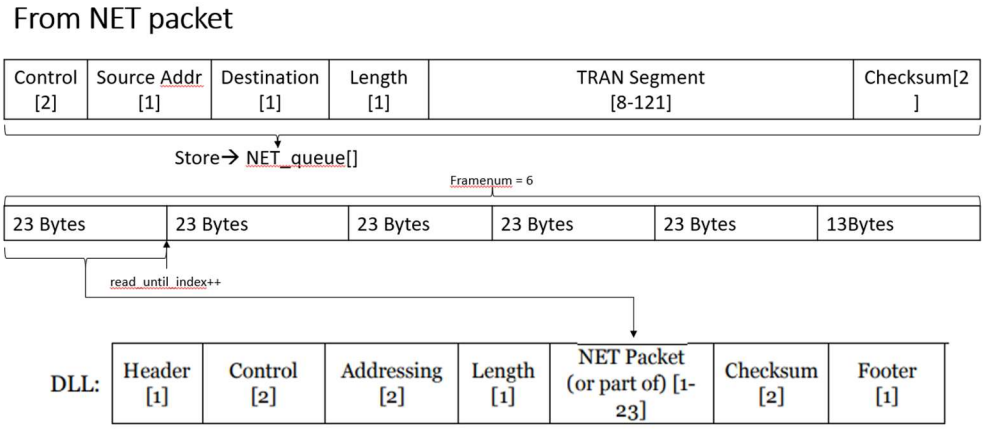

Figure 3 Breakdown of Network Layer payload

3.2.2. Flow Control

A simple handshake protocol is implemented on the system using two function: sender() and addrread(). On sending, the sender() function will be called and the sequence number increased by one. It will be sent to the destination address. On receiving, the addrread() function read the data from by calling member function from PHY layer, it decodes the payload and put in the packet data structure, once it completed, the address of destination and source is swapped and the Acknowledgement increase by one, so it sends back to the source. On source side, before getting the increased Acknowledgement signal, it will keep calling addrread() function before it sends out next frame.

3.2.3. Checksum

The checksum method used mainly is fletcher checksum. The cumulative sum of modulus 256 of each byte is put into A and the cumulative sum of A is assigned into B. The last byte of A and B

is then assigned to checksum in the DLL data packet. The member function errorchecking serves to verify the message match with the checksum bytes transmitted. On receiving, errorchecking() is called in addrread() so that if a packet passes the error checking only sends back acknowledgement.

3.3. Physical Layer

Class is used to build up the Physical Layer, the two member functions of physical layer txg_mg(Packet *message) and rx_msg() with a payload member use to store the packet received. In main function, the PHY layer is called in DLL to read the data. In rx_msg() function, the rx_poll() will be called and it reads the data continuously, however it will only stores the data into *messege[] buffer when it detects the header flag byte and stop taking in when it reads a footer byte.

4. Testing

In testing session, the program is not get tested since there is a compiler error in PC compiler, the cmd return stops thread shown in Figure 4. during makefile is called. Another alternative is using University Lab virtual machine to compile the file and sends the .hex back to Host PC. The cmd return a stop thread when loading in the hex file to ilmatto.

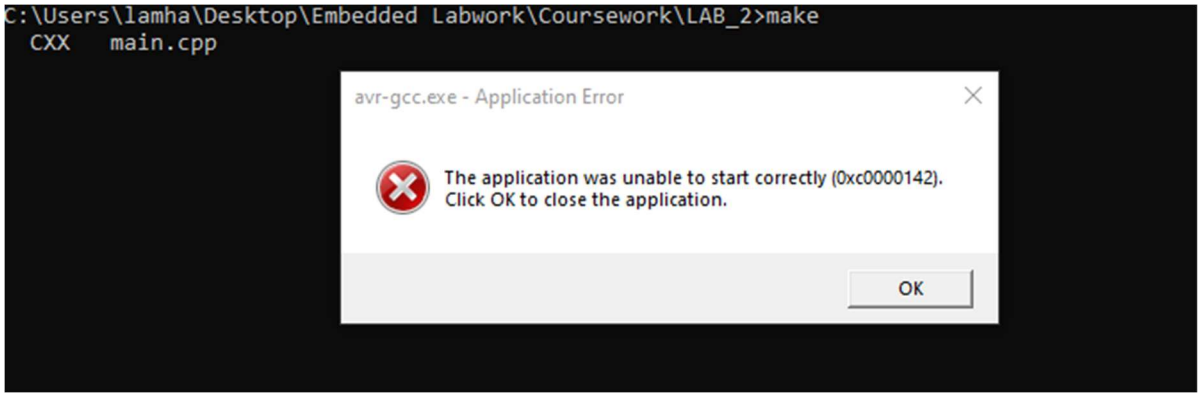

Figure 4 Unsolved compiler error

However, the checksum() and errorchecking() function is tested using G++ compiler. The result show in Figure 5 where the NET packet value is insert to the Payload packet and generate the checksum bytes using checksum() function. The NET packet is then insert back into error checking function. Result shown is correct, and result shown incorrect when payload function mutated.

```
C:\Users\lamha\Desktop\Embedded Labwork\Coursework\LAB_2>main
Net packt value: 74 64 1 f 79 5a 5a 5a 5a 45 5a 6b 5a 78 5a 3a 5a 5a 5a 5a 5a 7e 1
string length: 23
checksum bytes: 7a 1f
when Payload is used to check back itself:
result of errorchecking: 1
 Mutated Net packt value: 74 64 1 f 79 5a 5a 5a 1 99 5a 6b 5a 78 5a 3a 5a 5a 5a 5a 5a 7e 1
when Payload is mutated:
result of errorchecking: 0checksum bytes of mutated Net pkt: 75 80
C:\Users\lamha\Desktop\Embedded Labwork\Coursework\LAB_2>
```

Figure 5. Checksum testing

5. Critical Reflection and Evaluation

In this coursework, it is clearly that I have slow progression due to my late start. I would do start it early and brush up my C programming skill if I were given a chance. Also, one thing to mention about is that I did not completely implement the handshake protocol. The program will not resend a message when it does not receive an Acknowledgement from another node. Also, the 0-persistent broadcast is not implemented to the code. I should have communicated with my teammate when I encounter a compiler issue. During agreeing on the standard with peer teammate I should think from a user side rather than from designer side, this can raise more question to the standard, and also prevent unnecessary changes during design.

6. Reference

1. Tanenbaum, A. and Wetherall, D., 2010. *Computer Networks, Fifth Edition*. 5th ed. Prentice Hall, pp.193-251.

2. Hope RF, "Universal ISM Band FSK Transceiver module", RFM12B datasheet, Dec. 2006, [Accessed 08/11/2020], Available at https://cdn.sparkfun.com/datasheets/Wireless/General/RFM12B.pdf

Appendix

DLLnPHY.h

```
// Written by: Travis Lam Han Yuen
// Student ID: 30582105


//#include <avr/io.h>
#include <stdio.h>
#include <stdint.h>

#define NODE1 1
#define NODE2 5
#define NODE3 15
//#include the library for the RFM12 module and the UART
//#include "rfm12.h"

uint8_t node_addr = NODE1;
uint8_t HeaderFooter = 0x7E;
uint8_t flagbyte = 0x55;
uint8_t NET_packet[] = {
    0x74, 0x64,
    0x01, 0x0F,   // src and dest addr
    0x79,         // length
    0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A,
    0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A,
    0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A,
    0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A,
    0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A,
    0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A,
    0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A,
    0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A, 0x5A,
    0x5A, // Trans segment x121
    0x7E, 0x4C
};      // an example of NET packet


typedef struct Packet{
    uint8_t Header;  // 1 byte
    uint8_t Footer;  // 1 byte
    uint8_t Netpkt[];       // 1-23bytes
    uint8_t checksum[2];    // 2 bytes
    uint8_t Length;         // 1 byte
    uint8_t Address[2];     // 2 bytes
    uint8_t Control[2];     // 2 bytes
    uint8_t Everything[];   // bytes
} Packet;

class DLL {       // The class
    public:       // Access specifier
        DLL();    //constructor
        void from_NET_layer(uint8_t networkpayload[]);
```

```
    uint8_t Everything[];   // bytes
} Packet;

class DLL {       // The class
    public:       // Access specifier
        DLL();    //constructor
        void from_NET_layer(uint8_t networkpayload[]);
        Packet to_PHY_layer();
        uint8_t to_NET_layer();
    private:
        Packet Payload;
        void from_NET_layer(uint8_t networkpayload[]);
        Packet to_PHY_layer();
        uint8_t NET_queue_down[];
        uint8_t NET_queue_up[];
        uint16_t framenum;
        uint16_t read_until_index_down;
        uint16_t read_until_index_up;
        uint8_t Seq;
        uint8_t Ack;
        uint8_t CSmethod;

        /// on sending
        void putNET();
        void putleng();
        void addressing(uint8_t source,uint8_t destination);
        void checksum();
        void controlbytes();
        void sender();
        void encode_everything();

        /// on receiving
        bool errorchecking(Packet errorcheck);
        void addr_read();
        void decode_everything(Packet DL_packet_received);
};

class PHY {
    public:
        void from_DLL_layer(Packet payload);
        Packet to_DLL_layer();
    private:
        Packet* message;
        Packet Payload;
        void tx_msg(Packet* msg);
        void rx_msg();
};

void rx_poll(Packet* on_rx);
```

DLLnPHY.cpp

```
1  // Written by: Travis Lam Han Yuen
2  // Student ID: 30582105
3
4
5  #include "DLLnPHY.h"
6
7  DLL::DLL()
8  {
9      framenum = 0;
10     read_until_index_up = 0;
11     read_until_index_down = 0;
12     Seq = 0;
13     Ack = 0;
14     CSmethod =0;
15 }
16
17 void DLL::from_NET_layer(uint8_t networkpayload[])
18 {
19     uint16_t length =0;
20     while (NET_packet[length] != '\0')
21     {
22         length++;
23     }
24     for(uint8_t i = 0; i<length;i++)
25     {
26         NET_queue_down[i] = networkpayload[i];
27     }
28     addressing(NET_queue_down[2],NET_queue_down[3]);
29     // determine the frame number to be send, each frame of DLL has 32 bits.
30     framenum = length/32;
31     if((length%32)>0)
32     {
33         framenum+=1;
34     }
35     return;
36 }
37
38 Packet DLL::to_PHY_layer()
39 {
40     return Payload;
41 }
42
43 uint8_t DLL::to_NET_layer()
44 {
45     return NET_queue_up;
46 }
47 ////////////// on sending ///////////
48 void DLL::putNET()
49 {
```

```
    return;
}

Packet DLL::to_PHY_layer()
{
    return Payload;
}

uint8_t DLL::to_NET_layer()
{
    return NET_queue_up;
}
////////////// on sending ///////////
void DLL::putNET()
{
    for(uint8_t j = 0;j<23;j++)
    {
        Payload.Netpkt[j] = NET_queue_down[read_until_index];
        read_until_index++;
    }
    return;
}

void DLL::putleng()
{
    Payload.Length = (uint8_t) sizeof(Payload.Netpkt);
    return;
}

void DLL::addressing(uint8_t source,uint8_t destination)
{
    uint8_t addressbytes[2] = { 0x00,0x00};
    addressbytes[0] = source;
    addressbytes[1] = destination;
    Payload.Address[0] = addressbytes[0];
    Payload.Address[1] = addressbytes[1];
    return;
}

void DLL::checksum()
{
    uint16_t A[sizeof(Payload.Netpkt)];
    uint16_t B[sizeof(Payload.Netpkt)];
    uint16_t accumulate = 0;
    uint16_t accumulate_of_A = 0;
    for(uint8_t i=0;i<sizeof(Payload.Netpkt);i++)
    {
        accumulate += Payload.Netpkt[0];
        A[i] = accumulate;
```

```
74  void DLL::checksum()
75  {
76      uint16_t A[sizeof(Payload.Netpkt)];
77      uint16_t B[sizeof(Payload.Netpkt)];
78      uint16_t accumulate = 0;
79      uint16_t accumulate_of_A = 0;
80      for(uint8_t i=0;i<sizeof(Payload.Netpkt);i++)
81      {
82          accumulate += Payload.Netpkt[0];
83          A[i] = accumulate;
84          accumulate_of_A += A[i];
85          B[i] = accumulate_of_A;
86      }
87      Payload.checksum[0] = A[sizeof(Payload.Netpkt)-1]%256;
88      Payload.checksum[1] = B[sizeof(Payload.Netpkt)-1]%256; //  modulus 256 because 1 byte is
89      return;
90  }
91
92  Packet checksum(Packet Payload)
93  {
94      uint8_t length =23;
95      while(Payload.Netpkt[length] != 0)
96      {
97          length++;
98      }
99      cout<<dec <<(int)length << endl;
100     uint16_t A[length];
101     uint16_t B[length];
102     uint16_t accumulate = 0;
103     uint16_t accumulate_of_A = 0;
104     for(uint8_t i=0;i<length;i++)
105     {
106         accumulate += Payload.Netpkt[i];
107         A[i] = accumulate;
108         accumulate_of_A += A[i];
109         B[i] = accumulate_of_A;
110     }
111     Payload.checksum[0] = A[length-1]%256;
112     Payload.checksum[1] = B[length-1]%256; //  modulus 256 because 1 byte is 2**8 bits
113     return Payload;
114 }
115 void DLL::controlbytes()
116 {
117     Payload.Control[0] = (Seq<<4) | Ack;
118     Payload.Control[1] = (CSmethod<<4) | framenum;
119     return;
```

```
void DLL::encode_everything()
{
/*  Serialize-  1.Header,
                2.Control,
                3.Addressing,
                4.Length,
                5.Netpkt,
                6.Checksum,
                7.Footer,
*/
    putNET();
    uint16_t byte_count = 0;
    controlbytes();
    putleng();
    checksum();

    Payload.Header = 0x7E;
    Payload.Footer = 0x7E;

    Payload.Everything[byte_count] = Payload.Header;        ///////Header
    byte_count++;

    if(Payload.Control[0]==Payload.Header)
    {
        Payload.Everything[byte_count] = flagbyte;
        byte_count++;
    }

    Payload.Everything[byte_count] = Payload.Control[0];    /////////Control
    byte_count++;

    if(Payload.Control[1]==Payload.Header)
    {
        Payload.Everything[byte_count] = flagbyte;
        byte_count++;
    }

    Payload.Everything[byte_count] = Payload.Control[1];
    byte_count++;

    if(Payload.Address[0]==Payload.Header)
    {
        Payload.Everything[byte_count] = flagbyte;
        byte_count++;
    }

    Payload.Everything[byte_count] = Payload.Address[0];      ///////Address
    byte_count++;

```

```
        Payload.Everything[byte_count] = flagbyte;
        byte_count++;
    }

    Payload.Everything[byte_count] = Payload.Address[0];      ///////Address
    byte_count++;

    if(Payload.Address[1]==Payload.Header)
    {
        Payload.Everything[byte_count] = flagbyte;
        byte_count++;
    }

    Payload.Everything[byte_count] = Payload.Address[1];
    byte_count++;

    if(Payload.Length==Payload.Header)
    {
        Payload.Everything[byte_count] = flagbyte;
        byte_count++;
    }

    Payload.Everything[byte_count] = Payload.Length;       /////Length
    byte_count++;

    for(int x=0;x < sizeof(Payload.Netpkt);x++)
    {
        if(Payload.Netpkt[x]==Payload.Header)
        {
            Payload.Everything[byte_count] = flagbyte;
            byte_count++;
        }
        Payload.Everything[byte_count] = Payload.Netpkt[x];    ///// NET
        byte_count++;
    }

    if(Payload.checksum[0]==Payload.Header)
    {
        Payload.Everything[byte_count] = flagbyte;
        byte_count++;
    }

    Payload.Everything[byte_count] = Payload.checksum[0];      ///// Checksum
    byte_count++;

    if(Payload.checksum[1]==Payload.Header)
    {
        Payload.Everything[byte_count] = flagbyte;
        byte_count++;
```

```
204     }
205
206     Payload.Everything[byte_count] = Payload.checksum[0];      ///// Checksum
207     byte_count++;
208
209     if(Payload.checksum[1]==Payload.Header)
210     {
211         Payload.Everything[byte_count] = flagbyte;
212         byte_count++;
213     }
214
215     Payload.Everything[byte_count] = Payload.checksum[1];
216     byte_count++;
217
218     Payload.Everything[byte_count] = Payload.Footer;          ////// Footer
219     return;
220 }
221
222 void DLL::sender()
223 {
224     PHY phy_layer;
225     //from_NET_layer(NET_packet);
226     while(true)
227     {
228         from_NET_layer(NET_packet);
229         encode_everything();
230         phy_layer.from_DLL_layer(Payload);
231         Seq++;
232         uint8_t latch = Ack;
233         while(latch == Ack)
234         {
235             addr_read();
236             _delay_ms(50);
237         }
238     }
239     return;
240 }
241
242 ////////////////////// On receiving ////////////////
243
244 > void DLL::decode_everything(Packet DL_packet_received)···
355
356 > void DLL::addr_read()···
382
383 bool errorchecking(Packet errorcheck)
384 {
385     uint8_t length =23;
386     while(errorcheck.Netpkt[length] != 0)
387     {
```

```
244  void DLL::decode_everything(Packet DL_packet_received)
245  {
246      uint16_t byte_count1 = 0;
247      uint16_t byte_count2 = 0;
248      Payload.Header = DL_packet_received.Everything[byte_count2];
249      Payload.Everything[byte_count1] = DL_packet_received.Everything[byte_count2];
250      byte_count1++;
251      byte_count2++;
252
253      if(DL_packet_received.Everything[byte_count2] == flagbyte)
254      {
255          byte_count2++;    // see flag then skip
256      }
257
258      Payload.Control[0] = DL_packet_received.Everything[byte_count2];
259      Payload.Everything[byte_count1] = DL_packet_received.Everything[byte_count2];
260      byte_count1++;
261      byte_count2++;
262
263      if(DL_packet_received.Everything[byte_count2] == flagbyte)
264      {
265          byte_count2++;    // see flag then skip
266      }
267
268      Payload.Control[1] = DL_packet_received.Everything[byte_count2];
269      Payload.Everything[byte_count1] = DL_packet_received.Everything[byte_count2];
270      byte_count1++;
271      byte_count2++;
272
273      if(DL_packet_received.Everything[byte_count2] == flagbyte)
274      {
275          byte_count2++;    // see flag then skip
276      }
277
278      Payload.Address[0] = DL_packet_received.Everything[byte_count2];
279      Payload.Everything[byte_count1] = DL_packet_received.Everything[byte_count2];
280      byte_count1++;
281      byte_count2++;
282
283      if(DL_packet_received.Everything[byte_count2] == flagbyte)
284      {
285          byte_count2++;    // see flag then skip
286      }
287
288      Payload.Address[1] = DL_packet_received.Everything[byte_count2];
289      Payload.Everything[byte_count1] = DL_packet_received.Everything[byte_count2];
290      byte_count1++;
291      byte_count2++;
```

```
288     Payload.Address[1] = DL_packet_received.Everything[byte_count2];
289     Payload.Everything[byte_count1] = DL_packet_received.Everything[byte_count2];
290     byte_count1++;
291     byte_count2++;
292
293     if(DL_packet_received.Everything[byte_count2] == flagbyte)
294     {
295         byte_count2++;    // see flag then skip
296     }
297
298     Payload.Length = DL_packet_received.Everything[byte_count2];
299     Payload.Everything[byte_count1] = DL_packet_received.Everything[byte_count2];
300     byte_count1++;
301     byte_count2++;
302
303     for(uint8_t i=0;i<Payload.Length;i++)
304     {
305         if(DL_packet_received.Everything[byte_count2] == flagbyte)
306         {
307             byte_count2++;    // see flag then skip
308         }
309         Payload.Netpkt[i] = DL_packet_received.Everything[byte_count2];
310         NET_queue_up[read_until_index] = Payload.Netpkt[i];
311         read_until_index_up++;
312         framenum--;
313         if(framenum==0)
314         {
315             // send to NET layer
316         }
317         Payload.Everything[byte_count1] = DL_packet_received.Everything[byte_count2];
318         byte_count1++;
319         byte_count2++;
320     }
321
322     if(DL_packet_received.Everything[byte_count2] == flagbyte)
323     {
324         byte_count2++;    // see flag then skip
325     }
326
327     Payload.checksum[0] = DL_packet_received.Everything[byte_count2];
328     Payload.Everything[byte_count1] = DL_packet_received.Everything[byte_count2];
329     byte_count1++;
330     byte_count2++;
331
332     if(DL_packet_received.Everything[byte_count2] == flagbyte)
333     {
334         byte_count2++;    // see flag then skip
335     }
336
```

```
325         }
326
327         Payload.checksum[0] = DL_packet_received.Everything[byte_count2];
328         Payload.Everything[byte_count1] = DL_packet_received.Everything[byte_count2];
329         byte_count1++;
330         byte_count2++;
331
332         if(DL_packet_received.Everything[byte_count2] == flagbyte)
333         {
334             byte_count2++;    // see flag then skip
335         }
336
337         Payload.checksum[1] = DL_packet_received.Everything[byte_count2];
338         Payload.Everything[byte_count1] = DL_packet_received.Everything[byte_count2];
339         byte_count1++;
340         byte_count2++;
341
342         if(DL_packet_received.Everything[byte_count2] == flagbyte)
343         {
344             byte_count2++;    // see flag then skip
345         }
346
347         Payload.Footer = DL_packet_received.Everything[byte_count2];
348         Payload.Everything[byte_count1] = DL_packet_received.Everything[byte_count2];
349         byte_count1++;
350         byte_count2++;
351
352         return;
353
354     }
355
356     void DLL::addr_read()
357     {
358         PHY phy_layer;
359         decode_everything(phy_layer.to_DLL_layer());
360         if((Payload.Address[1]>>4) == node_addr)        //   read destination Address
361         {
362             uint8_t Ackbits = Payload.Control[0]<<4;
363             Ackbits = Payload.Control[0]>>4;
364             if(errorchecking(Payload))
365             {
366             ///  if errorchecking passed, Acknowledge bits +1 and send back to source
367                 if(Ackbits==15) ///  overflows set back to 0;
368                 {
369                     Payload.Control[0] = (Payload.Control[0]>>4);
370                     Payload.Control[0] = (Payload.Control[0]<<4);  // resets Ack
371                 }
372                 else
373                 {
```

```
372                 else
373                 {
374                     Payload.Control[0] += 1;
375                 }
376                 addressing(Payload.Address[1],Payload.Address[0]);  // swap the address and send agai
377                 phy_layer.from_DLL_layer(Payload);
378                 Ack++;
379             }
380         }
381     }
382
383     bool errorchecking(Packet errorcheck)
384     {
385         uint8_t length =23;
386         while(errorcheck.Netpkt[length] != 0)
387         {
388             length++;
389         }
390         cout<<dec <<(int)length << endl;
391         uint16_t A[length];
392         uint16_t B[length];
393         uint16_t accumulate = 0;
394         uint16_t accumulate_of_A = 0;
395         for(uint8_t i=0;i<length;i++)
396         {
397             accumulate += errorcheck.Netpkt[i];
398             A[i] = accumulate;
399             //cout <<dec << (int)A[i] << " ";
400             accumulate_of_A += A[i];
401             B[i] = accumulate_of_A;
402         }
403         if(errorcheck.checksum[0]==(A[length-1]%256) || errorcheck.checksum[1]==(B[length-1]%256))
404         {return true;}
405         else
406         {return false;}
407     }
408
409     ////////////////////////////////// PHY Layer /////////////////////////////////////
410
411     Packet PHY::to_DLL_layer()
412     {
413         rx_msg();
414         Packet tosend = *message;
415         return tosend;
416     }
417
418     void PHY::from_DLL_layer(Packet payload)
419     {
420         *message = Payload;
```

```
409 ////////////////////////////////// PHY Layer ////////////////////////////////////
410
411 Packet PHY::to_DLL_layer()
412 {
413     rx_msg();
414     Packet tosend = *message;
415     return tosend;
416 }
417
418 void PHY::from_DLL_layer(Packet payload)
419 {
420     *message = Payload;
421     tx_msg(message);
422     return;
423 }
424
425 void PHY::tx_msg(Packet *msg)
426 {
427     // // Determine the length of the string
428     // uint8_t length = 0;
429     // while (msg->Everything[length] != '\0')
430     // {
431     //  length++;
432     // }
433
434     // // Queue message for transmission on rmf12 module
435     // rfm12_tx(length, 0xEE, msg->Everything);
436
437     // // Tick the device to transmit
438     // rfm12_tick();
439 }
440
441 void PHY::rx_msg()
442 {
443     rx_poll(message);
444     return;
445 }
446
447 ////////////////////////////// non classified function ////////////////////////
448 void rx_poll(Packet *on_rx)
449 {
450     // if (rfm12_rx_status() == STATUS_COMPLETE)
451     // {
452     //  // Determine the length of the incoming data
453     //  uint8_t rx_length = rfm12_rx_len();
454     //     uint8_t *rx[100];
455
456     //  // Quick sanity check to ensure we are receiving good data
```

```
441  void PHY::rx_msg()
442  {
443      rx_poll(message);
444      return;
445  }
446
447  ////////////////////////////// non classified function ////////////////////////
448  void rx_poll(Packet *on_rx)
449  {
450      if (rfm12_rx_status() == STATUS_COMPLETE)
451      {
452          // Determine the length of the incoming data
453          uint8_t rx_length = rfm12_rx_len();
454          uint8_t *rx[100];
455
456          // Quick sanity check to ensure we are receiving good data
457          if ((rx_length == 0) || (rx_length > 100))
458          {
459              // Malformed data
460              return;
461          }
462
463          // Receive the data
464          memcpy(rx, rfm12_rx_buffer(), rx_length);
465          bool flag = false;
466          uint8_t x = 0;
467          for(uint8_t i = 0; i<sizeof(rx_length); i++)
468          {
469              if(*rx[i] == HeaderFooter){flag= !flag;}
470              while(flag == true)
471              {
472                  on_rx->Everything[x] = *rx[i];
473                  x++;
474                  if((*rx[i] == HeaderFooter)&&(*rx[i-1]!=flagbyte))
475                  {flag= !flag;}
476              }
477          }
478          // Clear the chip buffer after we read it
479          rfm12_rx_clear();
480
481      }
482  }
```

Mainfile

```
105  int main(void)
106  {
107      // init_uart0();    //init uart
108      // _delay_ms(100); //delay for the rfm12 to initialize properly
109      // rfm12_init();    //init the RFM12
110      // _delay_ms(100);
111      // sei(); //interrupts on
112
113      // while (1)
114      // {
115
116      // }
117  // only checksum and error checking function is tested //
118      uint8_t length =0;
119      Packet Payload;
120      Packet Mutated;
121      uint8_t values[24] = {0x74, 0x64,
122      0x01, 0x0F,   // src and dest addr
123      0x79,         // length
124      0x5A, 0x5A, 0x5A, 0x5A, 0x45, 0x5A, 0x6B, 0x5A, 0x78, 0x5A, 0x3A, 0x5A, 0x5A, 0x5A, 0x5A,
125      0x5A, 0x7E, 0x01, 0x00};
126      while(values[length] != 0)
127      {
128          length++;
129      }
130      memcpy(Payload.Netpkt, values, length);
131      cout<< "Net packt value: " ;
132      for(uint8_t i=0;i<length;i++)
133      {
134          cout << hex <<(int)Payload.Netpkt[i] << " ";
135      }
136      cout << endl<< "string length: "<<dec <<(int)length <<endl;
137      Payload = checksum(Payload);
138      cout <<"checksum bytes: ";
139      for(uint8_t i=0;i<2;i++)
140      {
141          cout << hex <<(int)Payload.checksum[i] << " " ;
142      }
143      cout << endl << "when Payload is used to check back itself:";
144      cout << endl<<"result of errorchecking: "<<errorchecking(Payload)<< endl;
145
146      uint8_t values2[24] = {0x74, 0x64,
147      0x01, 0x0F,   // src and dest addr
148      0x79,         // length
149      0x5A, 0x5A, 0x5A, 0x01, 0x99, 0x5A, 0x6B, 0x5A, 0x78, 0x5A, 0x3A, 0x5A, 0x5A, 0x5A, 0x5A,
150      0x5A, 0x7E, 0x01, 0x00};
151      while(values[length] != 0)
152      {
```

```
    Payload = checksum(Payload);
    cout <<"checksum bytes: ";
    for(uint8_t i=0;i<2;i++)
    {
        cout << hex <<(int)Payload.checksum[i] << " " ;
    }
    cout << endl << "when Payload is used to check back itself:";
    cout << endl<<"result of errorchecking: "<<errorchecking(Payload)<< endl;

    uint8_t values2[24] = {0x74, 0x64,
    0x01, 0x0F,   // src and dest addr
    0x79,         // length
    0x5A, 0x5A, 0x5A, 0x01, 0x99, 0x5A, 0x6B, 0x5A, 0x78, 0x5A, 0x3A, 0x5A, 0x5A, 0x5A, 0x5A,
    0x5A, 0x7E, 0x01, 0x00};
    while(values[length] != 0)
    {
        length++;
    }
    memcpy(Mutated.Netpkt, values2, length);
    cout<< " Mutated Net packt value: " ;
    for(uint8_t i=0;i<length;i++)
    {
        cout << hex <<(int)Mutated.Netpkt[i] << " ";
    }
    Mutated.checksum[0] = Payload.checksum[0];
    Mutated.checksum[1] = Payload.checksum[1];
    cout << endl << "when Payload is mutated:";
    cout << endl<<"result of errorchecking: "<<errorchecking(Mutated);
    Mutated = checksum(Mutated);
    cout <<"checksum bytes of mutated Net pkt: ";
    for(uint8_t i=0;i<2;i++)
    {
        cout << hex <<(int)Mutated.checksum[i] << " " ;
    }
    return 0;
}

// void uart_input(char
// {
//  // Show the user what they are typing
```