Do Wen Rei
29066832 (wrd1e16)
Team A2

1. Introduction

A simple network architecture that allow multiple users to chat together is implemented on II Matto and a RFM12B-S2 radio module. It aims to offer users to chat privately or broadcast the message with everyone. As Protocol layering method can be used to divide up network functionality, we referenced to the 5 layers model used in Tanenbaum as shown in Figure 1. I focused on Application and Transport Layer.

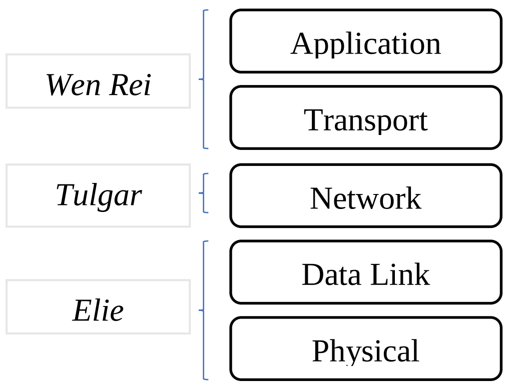

Figure 1 : 5 Layers Protocol referenced to Tanenbaum.

2.1 Standard Document for Transport Layer

Control [2 Bytes]

15 out of the available 16 control bits are used and the functionalities are shown as below.

Message/ack	Checksum 1	Checksum 0	Reliable?	Is fragment? (1-Fragment, 0 - Last Message/ No fragments)		Message/ACK data 1	Message/ACK data 0
Sequence no. [5-0]						Fragment no. [1-0]	

Table 1: Control Bits

The transport protocol is connectionless. Reliable message will be acknowledged by the recipient.

Message/Acknowledgement [1 Bit]

This bit is used to signal whether the packet is a message or acknowledgement, where 0 is message (Msg) and 1 is acknowledgement (Ack). It is presumed that messages will be received in order due to the nature of the lower layers.

Checksum [2 Bits]
Checksum control bits are used to offer four different types of checksum. The default 16-bit Interleaved Even Parity Checksum (00) is mandatory for compatibility. Other checksum types are optional. These are Internet RFC1071 Checksum (01), 16-bit Cyclic Redundancy Check (10) and 16-bit Fletcher Checksum (11).

Reliable [1 Bit]
Receiver will send an acknowledgement if a message is received with this bit is set to 1. During unreliable transmission (bit set to 0), receivers will not send an acknowledgment and the sender will not expect any. The receiver can choose to accept any checksum failures, attempt recovery, ignore fragmentation errors or simply drop the packet. Unreliable transmissions are mainly used broadcast messages.

Is Fragment [1 Bit]
This bit is used to indicate whether the packet is fragmented. The packet will be fragmented if the sent message is more than 114 bytes. If the packet is fragmented, this bit is set to 1 for all fragmented packets except the final fragment, which will be set to 0 to signify that it is the final fragment. When a packet with this bit unset is received the implementation should read the fragment number to determine whether more previous fragments are to be expected.

Message/Acknowledgement Status [2 Bits]
When the packet is an acknowledgement, these 2 bits will be filled with 00 (Success), 01 (Checksum fail), 10 (Missing fragment), 11 (No application listening on port). In the case of reliable transmission, receiver of acknowledgements will continue attempting retransmission when a received status is checksum fail or the missing fragment. The implementation may wish to retransmit immediately in these cases.

Sequence Number [6 Bits]
Sequence number increments with every sent message and the same sequence number is used in acknowledgement. This will ensure that the received acknowledgement is for the same sent packet. The sequence number will increase until $2^6= 64$ and loops back to 0.

Fragment Number [2 Bits]
Fragment number increment with every new fragment within the same sequence number. As the sent message is limited to 114 bytes per message as defined by the coursework, 2 bits of fragment number can increase the message length to $4 \times 114 = 456$ bytes to facilitate long messages. For each message in the sequence the fragment number will begin at 0. The fragment number of the final packet will indicate the total number of fragments, which can also be used to detect lost fragments.

SRC Port [1 Byte]

Signifies which port on the sender sent the message. Each application will choose a port from which to send messages. This will be passed to the application so that it may determine which port to return a message to if it wishes to reply.

DEST Port [1 Byte]

Signifies which port on the receiver is to receive the message. An application may listen on a specific port and will receive messages addressed to it.

Length [1 Byte]

The length of the data in each packet. In the case of multiple fragments, the receiver will sum the lengths for the total data size.

App Data [1 – 114 Bytes]

The actual message being sent by the application. The size of the message in each packet can vary between 1 to 114 bytes.

Checksum [2 Bytes]

Checksum of the type described in the checksum type control bits. Will check the whole transport packet with the initial checksum bits filled with 0s.

2.2 Standard Document for Application Layer

Applications will both listen on and send to port 21. Messages (up to the transport layer's maximum of 456 bytes) will be sent to the given ("to") address in plain text. The message text will be sent as the application data to the transport layer with no additional headers. Standard characters in the message will be encoded as ASCII.

3. Design

A simple command prompt interface is used for the application layer. Three distinct states are used to control the logic of the interface, which are Address, Message and Repeat. First, it asks for the destination address which will be accessed in network layer. Second, it prompts user to key in the message. Third, it lets user to continue talking to the same device or other devices.

In the transport layer, the source and destination ports are defined to be the same for simplification. It would work well as the IP address of each devices is different, which could be used for identification.

Two distinct functions are defined in the transport layer, which are transmit function and receive function. The sending device will call the transmit function where it will add header to the application message to form transport packets, which will be passed on to the network layer. The header is defined as in the standard document, which are the control bits, control and source ports, length of data and checksum. To ease the readability of the code, structures are used to define the header.

In the receiving device, the receive function will get the transport packets from the network layer and unpack it. It will first check the control bits and perform the subsequent operations. It will calculate the checksum based on the type of checksum chosen for the transport packet and compare with the received checksum bytes.

If the checksum fails and the transmission is reliable, it will attempt for retransmission through sending an acknowledgement packet with failed checksum status. However, if the transmission is unreliable, the receiver will ignore the failed checksum and display the wrong message.

Two types of checksum, which are Interleaved Even Bit Parity (00) and Fletcher checksum (11) are implemented.

4. Testing, Results and Analysis

To test the functioning of transport layer in isolation, loop back method is used where the output of the transmit function, which is the transport packet, is fed into the input of the receive function. In the debug mode, header from both the transmit and receive function is printed out to the UART interface in Hexadecimal for comparison as shown in Figure 2. If Reliable bit is set to 1, it will simulate reliable transmission where the receiver will send acknowledgements to the sender.

```
Welcome to A2 Il-Messenger
Which Device Do You Want to Talk To? [T,E,R]
T
This address is for Network Layer
Type In Your Message
Hello World!
Using Reliable Transmission -Expecting an Acknowledgement
Transport Segment in Hex
Src Port:15 Dest Port:15
Sequence Number:00 Fragment Number:00 Message Length (including newline):0D
Data: Hello World!
Checksum: 03 12
-----Sending packet through default physical layer -----
Receive Segment
Src Port:15 Dest Port:15
Sequence Number:00 Fragment Number:00 Message Length (including newline):0D
Data: Hello World!
Checksum: 03 12
Send Acknowledgement :
90 00 15 15 0D 88 15
```

Figure 2: Reliable Transmission in Debug Mode.

If Fletcher bit is set to 1, it will use Fletcher Checksum, otherwise it will be defaulted to Interleaved Even Parity Checksum. As these two are error detection checksums, retransmissions are required any error occurred.

```
Welcome to A2 Il-Messenger
Which Device Do You Want to Talk To? [T,E,R]
R
This address is for Network Layer
Type In Your Message
Hello World!
Using Fletcher Checksum
Using Reliable Transmission -Expecting an Acknowledgement
Transport Segment in Hex
Src Port:15 Dest Port:15
Sequence Number:00 Fragment Number:00 Message Length (including newline):0D
Data: Hello World!
Checksum: 63 12
-----Sending packet through default physical layer -----
```

Figure 3: Unreliable Transmission using Fletcher Checksum.

To simulate different scenarios in software within a device, preprocessor macros and if loops are used. If AddBitError is set to 1, the first data byte is purposely corrupted by replacing the first character with the second character using macros. Thus, the receiver will receive a corrupted message and the checksum will be incorrect, shown in Figure 4.

```
Welcome to A2 Il-Messenger
Which Device Do You Want to Talk To? [T,E,R]
E
This address is for Network Layer
Type In Your Message
Hello World!
Using Reliable Transmission -Expecting an Acknowledgement
Transport Segment in Hex
Src Port:15 Dest Port:15
Sequence Number:00 Fragment Number:00 Message Length (including newline):0D
Data: Hello World!
Checksum: 03 12
-----Sending packet through default physical layer -----
Receive Segment
Src Port:15 Dest Port:15
Sequence Number:00 Fragment Number:00 Message Length (including newline):0D
Data: eello World!
Checksum: 03 3F
Checksum Error
Checksum Error
Send Acknowledgement :
91 00 15 15 0D 89 15
Retransmitting Message
Receive Segment
Src Port:15 Dest Port:15
Sequence Number:00 Fragment Number:00 Message Length (including newline):0D
Data: Hello World!
Checksum: 03 12
Send Acknowledgement :
90 00 15 15 0D 88 15
```

Figure 4: Checksum Error. Sender retransmit message.

Sequence Number increases with every sent message as shown in Figure 5. Also, when a message is more than 114 characters within the same message, the fragment number increases but not the sequence number. Also, the IsFragment control bit is set to 1 to signal that it is a fragmented message as shown in Figure 6.

```
Type In Your Message
Hello World!
Using UnReliable Transmission -Not Expecting an Acknowledgement
Transport Segment in Hex
Src Port:15 Dest Port:15
Sequence Number:00 Fragment Number:00 Message Length (including newline):0D
Data: Hello World!
Checksum: 13 12
-----Sending packet through default physical layer -----
Do you still want to talk to this device? [Y,N]
Y
Type In Your Message
Nice to meet you!
Using UnReliable Transmission -Not Expecting an Acknowledgement
Transport Segment in Hex
Src Port:15 Dest Port:15
Sequence Number:01 Fragment Number:00 Message Length (including newline):12
Data: Nice to meet you!
Checksum: 4F 15
-----Sending packet through default physical layer -----
```

Figure 5: Increased Sequence Number.

```
Type In Your Message
0Yip2gCXtFGxkEPXTDI8cJvlDJf623j45iGqeJFXD0kSLKXw6GoA2FezYvDlflLRlooJqSHWmOOCiQ1Y
JywoME1zRPPvfkYAJFEThe00QIfa
 Fragmenting Message
jjy
Using UnReliable Transmission -Not Expecting an Acknowledgement
Transport Segment in Hex
Src Port:15 Dest Port:15
Sequence Number:03 Fragment Number:00 Message Length (including newline):71
Data: 0Yip2gCXtFGxkEPXTDI8cJvlDJf623j45iGqeJFXD0kSLKXw6GoA2FezYvDlflLRlooJqSHWmO
OCiQ1YJywoME1zRPPvfkYAJFEThe00QIfa
Checksum: 50 4D
-----Sending packet through default physical layer -----
Using UnReliable Transmission -Not Expecting an Acknowledgement
Transport Segment in Hex
Src Port:15 Dest Port:15
Sequence Number:03 Fragment Number:01 Message Length (including newline):04
Data: jjy
Checksum: 7E 0B
```

Figure 5: Increased Fragment Number when message is more than 114 characters.

5. Critical Reflection and Evaluation

Although our group tested on the minimal codes given during the lab sessions for transmitting and receiving messages, we did not manage to make a full and working 5 layers network architecture due to lack of coherence and time management. We tried to merge our codes, but we didn't have time to fully debug the errors. I believed we could do better if we set and adhere to a timeline and constantly meet-up to update on our progress. Also, we should all work on a brief pseudo-code to agree on the flow of the codes before diving deeper to develop on our own layer. Next, we should clearly define the function parameters for each layer and global variables.

The main weakness of the transport layer is that missing packets is not simulated as I did not get the AVR timer to work properly. Alternatively, I could implement counter at the sender side. Extra features such as crash recovery, flow control are not attempted. Also, the codes can be cleaner by defining multiple bit manipulation functions such as setting and clearing a bit from a byte as these operations are called multiple times.

I have no problem agreeing on the standard with my peer-teammates, Callum, as we discussed early and listen to each other's opinion. We drafted our transport and application standards before we started coding and it comes in handy during coding. Also, I have tested the control bits and results of the checksum with my peer-teammates.

Appendix

```
/** \file main.c
 * \author Wen Rei
 * \version 1.0
 * \date 20181210
 * \ Adapted from Domenico Balsamo's Code
 */

//#pragma GCC diagmostic ignored "-Wwrite-strings"
#include <avr/io.h>
#include <stdio.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include <string.h>
#include "rfm12.h"
#include "trans3.cpp"


#define SRC_PORT 21
#define DEST_PORT 21
#define TRANS_MAX 121
#define APP_MAX 114

uint8_t str[APP_MAX];   //the string want to send, taking each word as 1 byte
uint8_t str_2[APP_MAX]; //for fragmentation
char dest;   // Address for Network Layer
char c; // read uart char
uint8_t countchar=0; // count char in uart
uint8_t isframe=0;
uint8_t *bufptr;

int main(void)
{

    init_uart0();   //Initializa the uart
    _delay_ms(100);  //little delay for the rfm12 to initialize properly

    rfm12_init();    //init the RFM12
    _delay_ms(100);
    sei();           //interrupts on
    put_str("Welcome to A2 I1-Messenger\n\r");
    while(1){

    ADDRESS :
    put_str("Which Device Do You Want to Talk To? [T,E,R] \n\r");
        while(1)
    {
        //Get Destination Address
        if ((c = uart0_read()) != -1) {
            dest = c;
            put_ch((char) dest);
            put_str("\n\r");
            #if DEBUG
            put_str("This address is for Network Layer\n\r");
            #endif
            break;
        }
    }

    MESSAGE :
    put_str("Type In Your Message \n\r");
    while(1)
    {
        //If something is received through uart
        if ((c = uart0_read()) != -1) {
            str[countchar] = c;
            put_ch((char) str[countchar]);
            //count characters
            countchar++;
            //Segment Message if Message >114 characters
            if (countchar==APP_MAX-1){
                #if DEBUG
                put_str("\n\r Fragmenting Message \n\r");
                #endif
```

```
                isframe+=1;
                if(isframe==16)
                    isframe=0;
                memcpy(str_2,str,sizeof(str));
                *str = 0;
                countchar = 0;
            _delay_ms(500); //small delay so loop doesn't run as fast
            }
            //If newline :
            if(c == '\n' || c == '\r') {
                put_str("\n\r");
                // Transmit earlier fragmented message first
                if (isframe>=1){
                    _delay_us(500);
                    transmit_data(str_2,APP_MAX-1,isframe-1);
                    rfm12_tx(sizeof(str), 0, str); //it doesn't transmit, just buffe
                    for (uint8_t j = 0; j < 100; j++)
                    {
                    rfm12_tick();
                    _delay_us(500);
                    }
                    receive(transport_packet);
                }
                //Transmit Data
                transmit_data(str,countchar,isframe);
                // Send Message if New Line
                rfm12_tx(sizeof(str), 0, str); //it doesn't transmit, just buffer
                for (uint8_t j = 0; j < 100; j++)
                {
                rfm12_tick();
                _delay_us(500);
                }
                _delay_ms(500);
                *str = 0;
                countchar = 0;
                isframe=0;
                receive(transport_packet);
                break;
            }
        }
    }

    NEXT:
    put_str("\n\rDo you still want to talk to this device? [Y,N]\n\r");
    while(1)
    {
        //If something is received through uart
        if ((c = uart0_read()) != -1) {
            put_ch(c);

            if(c == 'Y' || c == 'Yes' ||c == 'y' ) {
                put_str("\n\r");
                goto MESSAGE; //MESSAGE
                break;
            }
            else if (c == 'N' || c == 'No' || 'n'){
                put_str("\n\r");
                goto RECEIVE;
                break;
            }
            else{
                put_str("\n\r Please type either Y or N");
                goto NEXT;
                break;
            }
        }
    }
    RECEIVE:
    if (rfm12_rx_status() == STATUS_COMPLETE)
        {
            put_str("rfm12_rx_status() = STATUS_COMPLETE \n\r");
            bufptr = rfm12_rx_buffer(); //get the address of the current rx buffer
            // dump buffer contents to uart
```

```
            for (uint8_t i=0;i<rfm12_rx_len();i++)
            {
                put_ch(bufptr[i]);
            }
                // tell the implementation that the buffer can be reused for the
                next data.
                rfm12_rx_clear();
                //_delay_ms(1000);
    }
    goto ADDRESS;
    }
}
```

```

//memcpy need this
#include<string.h>
#define SRC_PORT 21
#define DEST_PORT 21
#define APP_MAX 114
#define DEBUG 1
#define RELIABLE 1
#define ADDBITERROR 0
#define TRANS_MAX 121
#define FLETCHER 0
#define MISSING 0


void transmit_data(uint8_t* data,uint8_t length,uint8_t isframe);
void receive(uint8_t* net_packet);
uint16_t checksum(uint8_t*data,uint8_t length,uint8_t ctrl);
void sendAcknowledgement(uint8_t * trans_header);
void printByte(uint8_t byte);
char toCharInHex(uint8_t h_byte);
void put_strHex (uint8_t *str,uint8_t len);

```

```
#include "trans.h"
#define ACK_HEADER 7
uint8_t transport_packet[TRANS_MAX];
uint8_t trans_buffer[TRANS_MAX];
uint8_t seq=0x00;
uint8_t buffer[APP_MAX]; // buffer message for retransmission
uint8_t counterr=0x00;
struct trans{
    uint8_t ctrl[2];
    uint8_t src;
    uint8_t dest;
    uint8_t length;
    uint16_t checksum;

};

// Ctrl :0x00 Interleaved Parity, 0x60 Fletcher , 0x40 CRC, 0x20 Internet Checksum
uint16_t checksum(uint8_t* data,uint8_t length,uint8_t ctrl){
    switch (ctrl){
    case 0x00:{
    uint16_t parity=0x0000;
    for(int i=0;i<length;i++){
        if (i%2==0)
            parity^=(uint16_t)data[i]<<8;
        else
            parity^=data[i];
    }
    return parity;
    }

    case 0x01:{
    uint16_t sum_A = 0;
    uint16_t sum_B = 0;
   for(uint8_t i = 0; i < length; ++i )
   {
      sum_A = (sum_A + data[i]) % 255;
      sum_B = (sum_B + sum_A) % 255;
   }
   return (sum_B << 8) | sum_A;
    }
    }
}


void transmit_data(uint8_t* data,uint8_t length,uint8_t isframe){
    struct trans msg;
    uint8_t *t_ptr=transport_packet;
    uint8_t n;

    //First control bit
    msg.ctrl[0]=0x00; //[0Message,00Checksum,0Unreliable,0NoFragment,0Nth,00]
    #if FLETCHER
    put_str("Using Fletcher Checksum\n\r");
    msg.ctrl[0] |= (1<<5);
    msg.ctrl[0] |= (1<<6);
    #endif
    #if RELIABLE
    put_str("Using Reliable Transmission -Expecting an Acknowledgement \n\r");
    //set reliable bit
    msg.ctrl[0] |= (1<<4);
    memcpy(buffer,data,length);
    #endif
    #if !RELIABLE
    put_str("Using UnReliable Transmission -Not Expecting an Acknowledgement \n\r");
    #endif


    //Second control bit
    if(isframe !=0x00)
        msg.ctrl[0] |= 1<<3;
    msg.ctrl[1]=seq<<2 | (isframe); //[Sequence 6 bits, Fragment 2 bits]

```

```
    msg.src=SRC_PORT;
    msg.dest=DEST_PORT;
    msg.length=length;
    n=msg.length+7;

    //concatenate array for checksum, to checksum array varies according to the
    header_data size
    uint8_t to_checksum[n];
    to_checksum[0]=msg.ctrl[0];
    to_checksum[1]=msg.ctrl[1];
    to_checksum[2]=msg.src;
    to_checksum[3]=msg.dest;
    to_checksum[4]=msg.length;
    to_checksum[msg.length+5]=0x00;
    to_checksum[msg.length+6]=0x00;


    //concatenate header with data
    memcpy(to_checksum+5,data,msg.length);

    for(uint8_t i =0; i<sizeof(to_checksum);i++){
        printf("%x ",to_checksum[i]);
    }
    #if FLETCHER
    msg.checksum=checksum(to_checksum,sizeof(to_checksum),0x01);
    #endif

    msg.checksum=checksum(to_checksum,sizeof(to_checksum),0x00);
    printf("\nChecksum %x\n",msg.checksum);
    // MSB, LSB [masking to split msg.checksum to two 8 bits]
    uint8_t checksums[2]={ (uint8_t)(msg.checksum >> 8),(uint8_t)(msg.checksum &
    0xFF)};

    //to_checksum is now transport_packet after adding the checksum bits
    memcpy(to_checksum+msg.length+5,checksums,sizeof(checksums));

    put_str("Transport Segment in Hex \n\r");
    for(uint8_t i =0; i<sizeof(to_checksum);i++){
        printf("%x ",to_checksum[i]);
        *(t_ptr+i)=to_checksum[i];
    }
    //Increase seq number if it's not framed
    if(length!=0x71)
        seq+=1;
    if (seq==64)
        seq=0;
    #if ADDBITERROR
    memcpy(trans_buffer,t_ptr,sizeof(to_checksum));
    #endif
    #if DEBUG
    put_str("Src Port:");
    printByte(msg.src);
    put_str(" Dest Port:");
    printByte(msg.dest);
    put_str("\n\rSequence Number:");
    printByte(msg.ctrl[1]>>2);
    put_str(" Fragment Number:");
    printByte(msg.ctrl[1] & 0x03);
    put_str(" Message Length (including newline):");
    printByte(msg.length);
    put_str("\n\rData: ");
    put_strHex(data,msg.length);
    put_str("\n\rChecksum: ");
    printByte(to_checksum[msg.length+5]);
    put_ch(' ');
    printByte(to_checksum[msg.length+6]);
    put_str("\n\r");
    #endif

}

void receive(uint8_t* net_packet){
    trans* net=(trans*) net_packet;
```

```
    //save data
    uint8_t data[net->length];
    uint8_t data_length=net->length;
    #if ADDBITERROR
    counterr+=1;
    if (counterr==1)
    *(net_packet+5)=*(net_packet+6);
    #endif

    //GET DATA
    put_str("Receive Segment\n\r");
    for(uint8_t i =0; i<sizeof(data);i++){
        data[i]=*(net_packet+5+i);
        }
    //COMPUTE CHECKSUM
    uint8_t header_data[net->length+5];
    for(uint8_t i =0; i<sizeof(header_data);i++){
        header_data[i]=*(net_packet+i);
    }
    net->checksum=checksum(header_data,sizeof(header_data),net->ctrl[0] & 0x60);
    uint16_t given_checksum_r
    =((uint16_t)*(net_packet+5+net->length)<<8)|*(net_packet+5+net->length+1);


    #if DEBUG
    put_str("Src Port:");
    printByte(net->src);
    put_str(" Dest Port:");
    printByte(net->dest);
    put_str("\n\rSequence Number:");
    printByte(net->ctrl[1]>>2);
    put_str(" Fragment Number:");
    printByte(net->ctrl[1] & 0x03);
    put_str(" Message Length (including newline):");
    printByte(net->length);
    put_str("\n\rData: ");
    put_strHex(data,net->length);
    put_str("\n\rChecksum: ");
    printByte((uint8_t)(net->checksum >> 8));
    put_ch(' ');
    printByte((uint8_t)(net->checksum & 0xFF));
    put_str("\n\r");
    #endif
    if(net->checksum!=given_checksum_r){
        put_str("Checksum Error\n\r");
        }

    //If the communication is reliable, do checksum on original header data, print
    data to UART, set ack bit to 1 , do checksum for SendAcknowlendgement Bit
    if ( (net->ctrl[0] & (1<<4)) == 0x10){
        //put_str("\n\rReceive Block : Reliable");
        if(net->checksum!=given_checksum_r){
        net->ctrl[0] |=1<<0;
        put_str("Checksum Error\n\r");
        }
        net->ctrl[0] |= 1<<7;
        uint8_t trans_header[ACK_HEADER];
        trans_header[0]=net->ctrl[0];
        trans_header[1]=net->ctrl[1];
        trans_header[2]=net->src;
        trans_header[3]=net->dest;
        trans_header[4]=net->length;
        uint16_t ack_checksum=checksum(trans_header,ACK_HEADER-2,net->ctrl[0] & 0x60);
        trans_header[5]= (uint8_t)(ack_checksum >> 8);
        trans_header[6]=(uint8_t)(ack_checksum & 0xFF);
        sendAcknowledgement(trans_header);
    }
    else{

    }

```

```
void sendAcknowledgement(uint8_t * trans_header){
    put_str("Send Acknowledgement :\n\r");
    for(uint8_t i =0; i<7;i++){
        printByte(trans_header[i]);
        put_ch(' ');
        if(i==6)
            break;
    }
    put_str("\n\r");
    //Checksum Error - Do Retransmission
    if(trans_header[0] & (1<<0) ==0x01){
    put_str("Retransmitting Message\n\r");
    receive(trans_buffer);
    }

}

//Print the whole array (from hex to char)
void put_strHex (uint8_t *str,uint8_t len)
{
    int i;
    for (i=0; i<len; i++) {
        //printByte(str[i]);
        put_ch((char) str[i]);
    }
}


//Print a Byte in Hex
void printByte(uint8_t byte){
    uint8_t first_half=((0xF0 & byte)>>4);
    uint8_t second_half=(0x0F & byte);
    put_ch(toCharInHex(first_half));
    put_ch(toCharInHex(second_half));
}
//Convert Uint8_t to Char
char toCharInHex(uint8_t h_byte){
switch(h_byte){
    case 0x0:
    return '0';
    case 0x1:
    return '1';
    case 0x2:
    return '2';
    case 0x3:
    return '3';
    case 0x4:
    return '4';
    case 0x5:
    return '5';
    case 0x6:
    return '6';
    case 0x7:
    return '7';
    case 0x8:
    return '8';
    case 0x9:
    return '9';
    case 0xA:
    return 'A';
    case 0xB:
    return 'B';
    case 0xC:
    return 'C';
    case 0xD:
    return 'D';
    case 0xE:
    return 'E';
    case 0xF:
    return 'F';
}
```