Computer Networks

1st Project - Data Link Protocol

Gabriela Rodrigues da Silva - up202206777
Karolina Jedraszek - up202402265

Summary

Within the scope of the Computer Networks course, we have been asked to create an application capable of transmitting arrays of bytes - in the form of files - between a transmitter and a receiver through a serial port. To do so, we had to implement a Stop-and-Wait protocol, with well-defined frame structures and resilience to errors and timeouts.
Over the course of this project, we have developed a deeper understanding of the technologies involved in communicating between different machines, the related challenges, and how to overcome them. Moreover, we have explored the importance of subdividing a problem of large dimensions into manageable layers which build off each other's services.

Introduction

Communication across diverse computer systems presents a significant challenge and is of paramount importance. To address this, two layers of protocols have been studied and implemented: the Application Layer and the Data Link Layer. This project aims to explore and deepen understanding of these mechanisms through hands-on implementation, guided by the constraints of an established communication protocol and the limitations of a physical medium.
In this report, the results of this process are discussed and evaluated. It is divided into two main topics:

- The project's code, including:
    - **Architecture and Code Structure -** The code modules that constitute the application and their respective functions; implementation details for the most important functions;
    - **Main Use Cases -** How they are integrated into the larger program;
    - **Protocol Descriptions:**
        - **Logical Link Protocol**
        - **Application Protocol**
- The results of the program, namely:
    - **Validation and Efficiency -** How the correctness of the code was evaluated and quantified and statistics regarding the performance of the code.

Architecture And Code Structure

Internally, the project's code possesses four main modules:

- **main.c -** which was provided as a finished unit and serves as the startpoint of the program, calling, after minimal setup, the main application layer function, whose parameters are determined by the *argv* parameter supplied at initialization.
- **application_layer.h/.c -** which handles the higher-level Application Layer logic. It exposes the function:
    - **void applicationLayer(const char *serialPort, const char *role, int baudRate, int nTries, int timeout, const char *filename)** - responsible for the main functionality of the application: opening the files to be read from or written to, and calling the appropriate Data Link Layer functions in order to establish a connection, command the transmission or reception - depending on the role parameter - of packets in the appropriate order, and terminate according to the occurrence, or lack thereof, of errors.

Inaccessible to the rest of the application, auxiliary functions and structs were also created:

- **struct PointerIntPair -** used to associate the pointer to a block of bytes to the number of said bytes;
- **void readFileSize() -** uses calls to *fseek* and *ftell* in order to determine the size of the file indicated;
- **void splitFile() -** reads the next set of bytes from the file, storing them in global variables accessed in other functions;
- **PointerIntPair createDataPacket() -** allocates memory and fills it with a packet containing an header indicating order and size, and data from the file;
- **PointerIntPair createControlPacket(int option, const char *filename) -** allocates memory and fills it with a packet containing general information about the file, used to indicate the start and end of a transmission;
- **int parsePacket(unsigned char *packet, int size) -** used by the reader to determine which type of packet has been received, writing it to a file if necessary, or returning a termination code in case it is an end-of-transmission packet.

- **link_layer.h/.c -** which handles the lower-level Data Link Layer logic. It exposes the functions:
    - **int llopen(LinkLayer connection parameters) -** passes initialization information to the Physical Layer and proceeds with the establishment of a connection, adapting the process depending on the role(transmitter or received) assigned;
    - **int llwrite(const unsigned char *buf, int bufSize) -** generates the correct data frame structure from an array of bytes and executes a series of attempts to write a message until either the correct receiver ready response is received or the maximum number of attempts has been reached;
    - **int llread(unsigned char *packet) -** accumulates the message bytes received from the serial port. If the error correction bytes are not correct, a reject message is sent and another read attempt is started, up to a maximum number of times. In case a set or disconnect message is received, it will respond appropriately;

    - **int llclose(int showStatistics) -** used by the transmitter to send the disconnect frame and await a response from the receiver, up to a predetermined number of attempts. Prints, when the parameter is not 0, the number of data bytes sent with, and the number of calls to, llwrite.

  Two further functions were defined internally:
  - **Void alarmHandler(int signal) -** callback for the timeout alarm;
  - **int terminate_reader() -** as the call to llclose is exclusive to the transmitter, this function is called by llread upon receiving a disconnect message and handles the termination of the connection.
  - **int data_is_correct(unsigned char *data, unsigned int data_length, unsigned char bcc2) -** checks whether an array of bytes matches the given protection field value.
  -
- **serial_port.h/.c -** which was provided as a finished unit and functions as a minimal wrapper for the serial port initialization logic and read and write operations - the Physical Layer logic -, through the use of the functions:
  - **int openSerialPort(const char *serialPort, int baudRate) -** configures the serial port connection;
  - **int closeSerialPort() -** closes the serial port connection;
  - **int readByteSerialPort(unsigned char *byte) -** tries to read a byte from the serial port. As the read operation is non-blocking, it does not guarantee that a byte will be read, in which case the return value will be 0 instead of 1;
  - **Int writeBytesSerialPort(unsigned char *bytes, int numBytes) -** writes the supplied number of bytes to the serial port.

Main Use Cases

The program is prepared to adapt its behaviour according to parameters received in the main function and passed down to lower layers. It can function in two main modes:

- **Transmitter -** Which is responsible for splitting the file, and creating and sending data frames. A normal execution of the program as a transmitter obeys the following sequence of function calls:
  - application_layer()
    - llopen()
    - fopen()
    - readFileSize()
    - createControlPacket()
    - llwrite()
    - *While the end of the file has not been reached:*
      - createDataPacket()
      - llwrite()
    - createControlPacket
    - llwrite
    - fclose()

  If any of these steps returns an error code, the application terminates with exit(-1).

- **Receiver -** which is responsible for creating a new file, receiving and interpreting packets and filling in the file accordingly. A normal execution of the program as a receiver obeys the following sequence of function calls:
    - application_layer()
        - llopen()
        - fopen()
        - *While the Data Link Layer has not identified the disconnection message:*
            - *llread()*
            - *parsePacket()*
                - *If is a data packet:*
                    - fwrite()

  If any of these steps returns an error code, the application terminates with exit(-1). It can be noted that the receiver does not call llclose, as that function covers the transmitter's end of an asymmetrical exchange, while the receiver's is handled in llread directly.

Additionally, the baud rate, retry count, timeout duration, and the specific serial port designation can be customised.

Logical Link Protocol

The logical link protocol manages the data transmission between the transmitter and receiver at a lower level than the application layer.
We distinguish three phases: 1. establishing a connection (llopen), 2. transmitting data frames (llwrite, llread), 3. termination (llclose).
Each phase uses specific frames to transmit information:

1. **SET** sent by the transmitter to initiate a connection and **UA** sent back as a confirmation by a receiver
2. **Frames** created out of **data** packets sent by a transmitter, **RR0** / **RR1** sent back by the receiver that it is ready to receive an information frame number 0/1 or **REJ0** / **REJ1** sent by the receiver that it rejects an information frame number 0/1.
3. **DISC** sent first by the transmitter, then by a receiver and **UA** sent back by the transmitter.

We use numbers 0 / 1 in rejection and "ready to receive" frames to make sure the frames are appearing in the correct order.
In llwrite function, while creating frames to be sent, **stuffing** is performed. If a FLAG or ESCAPE byte occurs, it is changed to 0x7d 0x5e, this is why the number of bytes can be increased and bcc2, which is a protection field, transformed.

```
    if (buf[i] == ESCAPE || buf[i] == FLAG)
      {
          to_send[num_bytes] = ESCAPE;
          num_bytes++;
          to_send[num_bytes] = buf[i] ^ SPECIAL_MASK;
          num_bytes++;
      }
```

The llread function evokes the terminate_reader function. It, alongside llwrite, llopen and llclose is a **state machine** that has an internal while loop, running until either a successful communication occurs or the maximum number of attempts MAX_ALARM_REPEATS has not been reached.
Moreover, the TIMEOUT variable controls how long to wait for an acknowledgment after sending a frame. After each frame transmission, a timer is set based on its value. If no acknowledgment is received during this period, the timer expires, a retransmission is made.
To manage this process the alarmHandler function is implemented:

```
void alarmHandler(int signal)
{
   alarmEnabled = FALSE;
   alarm(0);

   alarmCount++;

   printf("Alarm #%d\n", alarmCount);
}
```

Application Protocol

Main goal of the application protocol is to manage the high-level process of transferring a file from a transmitter to a receiver, using the lower-level link layer functions for actual data transmission.
Information in the application protocol is passed to llwrite in **packets**. There are two main types of packets: control packets and data packets. Control packets are either start packets or end packets.
Information received by the application protocol from llread is stored in the buffer receivedbuf.

```
#define RECEIVE_BUFFER_SIZE 2048
[...]
unsigned char receivedbuf[RECEIVE_BUFFER_SIZE];
```

Beneath, the steps taken by the protocol are presented. Firstly, however, regardless of the role, it initialises the link layer connection, calling llopen. Depending on the chosen role (transmitter / receiver) it follows listed steps.

- **Transmitter** - creates packets (out of the file to be sent) and sends them using the data link layer. Opens the file to be transmitted, saves its name (filename) and calls a function readFileSize to save the size (filesize) as well.
    1. Creates a start control packet by calling the createControlPacket(0,filename) function. The created control packet is sent through llwrite function.
    2. The transmitter enters a loop to read and send parts of the file in parts of size **SEND_BUFFER_SIZE = 16 bytes**. In each loop step, it creates data packets (using createDataPacket) and calls llwrite() to transmit them. The loop ends when all bytes of the file are read and transmitted.
    3. The end control packet is created (createControlPacket(1,filename)) and sent through llwrite().
    4. The file is closed and the code calls llclose(1) to terminate the link layer connection. If any problems occur in llclose, it prints the error message and exits.
- **Receiver** - receives and interprets packets from the transmitter in order to reconstruct and save the file.
    1. Opens a new file (in the binary write mode “wb”, with the provided name = filename) to store the incoming data.
    2. It loops and listens continuously for the incoming packets by calling llread. It stores them in res. In order to know what type of packet is received the parsePacket() function is called. The function processes packets based on their types, which are determined by examining the control byte.
        - start control packet (packet[0] == 1) - parsePacket() extracts metadata like the file name and size, which allows the receiver to verify that the correct file is being received.
        - data packet (packet[0] == 2) - the receiver validates the packet sequence to ensure that packets arrive in order. Valid packets are parsed to extract and write the file data to the open file, if packets are too short or too long an error is printed.

```
if (size > SEND_BUFFER_SIZE + NUM_HEADER_BYTES)
 {
     printf("Too large packet received(size=%u)\n",size);
     return -2;
 }
```

        - end control packet (packet[0] == 3) - the receiver verifies that this packet matches the metadata from the start packet.

    The parsePacket() function returns an int (-1, -2,-3,-4) for errors and (1,2,3) for start packet, data packet and end packet. If the end packet is received, the **llread function is called**, which then **evokes terminate_reader function** to close the connection. The result of llread is stored in res_ variable. If errors occur, the respective message is printed by the application layer. If not, the loop ends.

```
int res_ = llread(receivedbuf);
                    if (res_ == -2)
                    {
                        printf("should end correctly!\n");
                        end = TRUE;
                    }
                    else if (res_ == -3)
                    {
                        printf("terminated incorrectly!!!\n");
                        exit(-1);
                        break;
                    }
```

Moreover, the usleep function was used in this main loop for the testing stage of the program. However, the SLEEP_AMOUNT variable was now set to zero, as introducing delay would not have reduced the number of errors.

```
#define SLEEP_AMOUNT 0
[...]
usleep(SLEEP_AMOUNT);
```

Validation And Data Link Protocol Efficiency

To verify whether the program transferred the file without errors, the contents of the initial and final files were compared with the diff command. For a bit error ratio of 0, they were consistently identical. With a higher bit error ratio, the program behaves stably. It was designed with the goal of terminating upon error detection, rather than creating a wrong file, and a low packet size of 16 bytes was chosen to minimise the probability of errors.
To evaluate the efficiency of the program, it was run repeatedly for the penguin.gif file using the bash scripts present in the annex of this report. An average of the results was then performed, as observable in the table below.

		Bit Error Ratio			
		0.0	0.0001	0.0005	0.001
Baud	1200	System execution time: 2m40.763s	System execution time: 4m15.638s	Unfeasible to calculate accurately due to execution time.	Unfeasible to calculate accurately due to execution time.
	9600	System execution time: 22.883s	System execution time: 1m59.547s	System execution time:8m02.058s	Unfeasible to calculate accurately due to execution time.
	115200	System execution time: 3.346s	System execution time: 1m27.178s	System execution time: 7m2.339s Rate of failure: 30%	System execution time: 13m58.278s Rate of failure: 30%
Rate of failure		0%	0%	15%	30%

Given that the Stop And Wait protocol was used, and that this protocol does not use the bandwidth available to its full extent due to waiting times between frame transmission and the reception of the acknowledgment, it is interesting to compare how much, exactly, the intervals of time obtained compare to the theoretical minimum.
To start, the protocol is transfering 17978 unique bytes (including headers), split between roughly 562 frames, adding up to a total of 143824 bits. Secondly, the baud rate indicates the amount of symbols (bits, in this case) transferred per unit of time. Therefore, optimally, for a baud rate of 9600, only 14.982 seconds would be needed to transfer the file, which is lower than the 22.883s obtained (the former is 65.47% of the latter). For baud rates of 1200 and 115200, this percentage is, respectively, 74.55% and 37.31%. The efficiency lowers as the baud rate increases because the propagation time remains constant, making it proportionally larger compared to the frame transmission time.

Conclusions

While working on this project, we gained hands-on experience in implementing a Stop-and-Wait protocol for a file transmission over a serial port. By developing structured layers - the data link layer and the application layer, we managed to achieve a reliable data transfer that handles errors and performs checks for data integrity.

The logical link protocol manages the data transmission between the transmitter and receiver at a lower level using frames. On the other hand, the application layer splits files into packets and coordinates their transmission using functions from the logical link protocol.

Techniques like byte stuffing were used to prevent data from containing premature end flags, while automatic retransmissions for unacknowledged frames addressed issues of data loss and corruption. Specifically, functions used to transfer frames within the data link layer incorporated state machines and alarms to manage retransmissions.

The main focus was to ensure that files were accurately received and processed. Testing across different bit error rates and baud rates illustrated both the strengths and limitations of our implementation of the Stop-and-Wait protocol. Particularly, how it balances reliability and efficiency in environments with higher error rates.

Overall, this project gave us a firmer understanding of network protocol design, layered architecture, and effective error management techniques.

Appendix I - Source Code

- application_layer.c

```
// Application layer protocol implementation

#include "application_layer.h"
#include "link_layer.h"

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>

#define MIN(x,y) ((x)<(y)?(x):(y))

#define SLEEP_AMOUNT 0
#define SEND_BUFFER_SIZE 16
// do not change the num header bytes should be 4
#define NUM_HEADER_BYTES 4
#define RECEIVE_BUFFER_SIZE 2048
unsigned char buf[SEND_BUFFER_SIZE];
unsigned char receivedbuf[RECEIVE_BUFFER_SIZE]; // could be more
int bytes;
int bufindex = 0;
int S = 0; // number of current packet
FILE *fptr;
long filesize;

typedef struct
{
    unsigned char *pointer;
    int size;
} PointerIntPair;

void readFileSize()
{
    fseek(fptr, 0, SEEK_END);
    filesize = ftell(fptr);
    fseek(fptr, 0, SEEK_SET);
}
```

```
void splitFile()
{
    bytes = fread(buf, 1, SEND_BUFFER_SIZE, fptr);
}

PointerIntPair createDataPacket() // returns data packet
{
    S++;
    printf("packet number: %d (as a byte:%d)\n", S, S % 256);
    unsigned char *datapacket = (unsigned char *)malloc((bytes +
NUM_HEADER_BYTES) * sizeof(unsigned char));
    datapacket[0] = 2;
    datapacket[1] = S;
    datapacket[2] = bytes / 256; // L2
    datapacket[3] = bytes % 256; // L3

    for (int i = 0; i < bytes; i++)
    {
        datapacket[4 + i] = buf[i];
    }

    PointerIntPair result;
    result.pointer = datapacket;
    result.size = 4 + bytes;
    return result;
}

PointerIntPair createControlPacket(int option, const char *filename) //
option is 0 for start packet 1 for end packet
{
    unsigned char lenfilename = (unsigned char)strlen(filename);
    unsigned char *controlpacket = (unsigned char *)malloc((13 +
lenfilename) * sizeof(unsigned char));
    if (option == 0) // start control packet
    {
        controlpacket[0] = 1;
    }
    else
        controlpacket[0] = 3;

    controlpacket[1] = 0; // file size
    controlpacket[2] = 8; // bytes of length (filesize)
    int tmp = 56;
    for (int i = 0; i < 8; i++)
    {
        controlpacket[i + 3] = (filesize >> tmp) & 0xFF;
```

```
        tmp -= 8;
    }
    controlpacket[11] = 1; // file name
    controlpacket[12] = lenfilename;
    memcpy(&controlpacket[13], filename, lenfilename);

    PointerIntPair result;
    result.pointer = controlpacket;
    result.size = 13 + lenfilename;

    return result;
}

int parsePacket(unsigned char *packet, int size)
{
    const static int TOO_SHORT_MAX_NUMBER = 4;
    static int too_short_counter = FALSE;
    static unsigned char lastPacketValue = 0;
    if (packet[0] == 1)
    {
        return 1;
    }
    else if (packet[0] == 2)
    {
        unsigned char current_packet = packet[1];
        if (size > SEND_BUFFER_SIZE + NUM_HEADER_BYTES)
        {
            printf("Too large packet received(size=%u)\n",size);
            return -2;
        }

        if ((unsigned char)current_packet != (unsigned
char)(lastPacketValue + 1))
        {
            printf("Out of order(previous %u vs current %u) \n",
lastPacketValue, current_packet);
            return -1;
        }

        // TODO: remove, maybe!!!
        if (size < SEND_BUFFER_SIZE + NUM_HEADER_BYTES)
        {
            if (too_short_counter < TOO_SHORT_MAX_NUMBER)
            {
                too_short_counter += 1;
                printf("Too short frame received!!!\n");
```

```
            }
            else
            {
                printf("Too many short frames in a row! This doesn't
make sense.\n\n");
                exit(-1);
            }
        }
        else
        {
            too_short_counter = 0;
        }

        lastPacketValue = current_packet;

        int L1 = packet[3];
        int L2 = packet[2];
        fwrite(&packet[4], 1, MIN(L2 * 256 + L1,SEND_BUFFER_SIZE),
fptr);
        printf("packet number: %u \n\n", packet[1]);
        return 2;
    }
    else if (packet[0] == 3)
        return 3;

    return 4;
}

void applicationLayer(const char *serialPort, const char *role, int
baudRate,
                      int nTries, int timeout, const char *filename)
{
    LinkLayer connectionParameters;
    strcpy(connectionParameters.serialPort, serialPort);
    connectionParameters.baudRate = baudRate;
    connectionParameters.nRetransmissions = nTries;
    connectionParameters.timeout = timeout;
    if (strcmp(role, "tx") == 0)
        connectionParameters.role = LlTx;
    else if (strcmp(role, "rx") == 0)
        connectionParameters.role = LlRx;

    // TODO: for safety, check if end info packet has the same
information as the start info packet

    printf("llopen try loop called\n");
```

```
    if (llopen(connectionParameters) < 0)
    {
        exit(-1);
    }

    printf("connection established.\n\n");

    if (strcmp(role, "tx") == 0) // transmiter
    {
        fptr = fopen(filename, "rb");
        readFileSize();

        // sedning start control packet
        PointerIntPair controlpacketstart = createControlPacket(0,
filename);

        int res;
        if (llwrite(controlpacketstart.pointer, controlpacketstart.size)
< 0)
        {

            printf("Error in writing start control packet\n");
            exit(-1);
        }

        free(controlpacketstart.pointer);

        // sending data packets
        do
        {
            splitFile();
            PointerIntPair datapacket = createDataPacket();

            res = llwrite(datapacket.pointer, datapacket.size);
            if (res == -1)
            {
                printf("Error in llwrite\n");
                exit(-1);
            }
            else if (res == -2)
            { // the dirtiest thing so far in this code :/
                printf("Error: unexpected RR or REJ code -> skipping to
next packet\n\n");
            }
            else if (res == -3)
```

```
            {
                printf("Randomly dissapearing bytes error :)\n");
                exit(-1);
            }

            usleep(SLEEP_AMOUNT);
            free(datapacket.pointer);
        } while (bytes > 0);

        // sending end control packet
        PointerIntPair controlpacketend = createControlPacket(1,
filename);

        if (llwrite(controlpacketend.pointer, controlpacketend.size) <
0)
        {
            printf("Error in writing end control packet\n");
            exit(-1);
        }

        free(controlpacketend.pointer);
        fclose(fptr);

        if (llclose(1) < 0)
        {
            printf("Error in llclose.\n");
            exit(-1);
        }
    }
    else if (strcmp(role, "rx") == 0) // receiver
    {
        fptr = fopen(filename, "wb");
        int end = FALSE;
        while (!end)
        {

            int res = llread(receivedbuf);
            printf("bytes received: %d\n", res);
            switch (res)
            {
            case -2:
                printf("Terminated correctly.\n");
                end = TRUE;
                break;
            case -3:
                printf("terminated with errors!\n");
```

```
            break;
        case -4:
            printf("SET received: resetting the file.\n");
            rewind(fptr);
            break;
        case -1:
            printf("Error in llread.\n");
            exit(-1);
            break;
        default: // should be correct
        {
            int resParse = parsePacket(receivedbuf, res);
            if (resParse == -1)
            {
                printf("Critical frame order error!\n");
                exit(-1);
                break;
            }
              if (resParse == -2)
            {
                printf("Assumed everything is fine\n");
                break;
            }

            if (resParse == 3)
            {

                printf("end packet received\n");

                int res_ = llread(receivedbuf);
                if (res_ == -2)
                {
                    printf("should end correctly!\n");
                    end = TRUE;
                }
                else if (res_ == -3)
                {
                    printf("terminated incorrectly!!!\n");
                    exit(-1);
                    break;
                }
            }
        }
        break;
        }
```

```
            usleep(SLEEP_AMOUNT);
        }
        printf("llread ended\n");
        fclose(fptr);
    }
    printf("Terminating application layer!\n");
}
```

- link_layer.c

```
// Link layer protocol implementation

#include "link_layer.h"
#include "serial_port.h"
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
// MISC
#define _POSIX_SOURCE 1 // POSIX compliant source

static int frame_num = 0;

static unsigned int errors_read = 0;
static unsigned int bytes_sent = 0;
static unsigned int swrite_calls = 0;
static unsigned int actual_bytes_sent = 0;

#define FALSE 0
#define TRUE 1

#define FLAG 0x7E
#define CTRL_SET 0x03
#define CTRL_UA 0x07
#define CTRL_I1 0x80
#define CTRL_I0 0x00
#define CTRL_RR0 0xAA
#define CTRL_RR1 0xAB
#define CTRL_REJ0 0x54
#define CTRL_REJ1 0x55
#define CTRL_DISC 0x0B
#define ADDR_SX 0x03
#define ADDR_RX 0x01

int TIMEOUT = 5;
```

```
int MAX_ALARM_REPEATS = 5;

#define TRIES 10
#define RR_LOST_TRIES 2

#define ESCAPE 0x7D
#define SPECIAL_MASK 0x20
#define LLWRITE_EXTRA_BIT_NUM 8

int alarmEnabled = FALSE;
int alarmCount = 0;

// Alarm function handler
void alarmHandler(int signal)
{
    alarmEnabled = FALSE;
    alarm(0);

    alarmCount++;

    printf("Alarm #%d\n", alarmCount);
}

////////////////////////////////////////////////
// LLOPEN
////////////////////////////////////////////////

enum OPEN_STATE
{
    STATE_START = 0,
    STATE_FLAG_RCV = 1,
    STATE_A_RCV = 2,
    STATE_C_RCV = 3,
    STATE_BCC_OK = 4,
};
#define SHORT_MESSAGE_SIZE 5
const unsigned char SET[] = {FLAG, ADDR_SX, CTRL_SET, ADDR_SX ^
CTRL_SET, FLAG};
const unsigned char UA[] = {FLAG, ADDR_SX, CTRL_UA, ADDR_SX ^ CTRL_UA,
FLAG};
const unsigned char RR0[] = {FLAG, ADDR_SX, CTRL_RR0, ADDR_SX ^
CTRL_RR0, FLAG};
const unsigned char RR1[] = {FLAG, ADDR_SX, CTRL_RR1, ADDR_SX ^
CTRL_RR1, FLAG};
const unsigned char REJ0[] = {FLAG, ADDR_SX, CTRL_REJ0, ADDR_SX ^
CTRL_REJ0, FLAG};
```

```
const unsigned char REJ1[] = {FLAG, ADDR_SX, CTRL_REJ1, ADDR_SX ^
CTRL_REJ1, FLAG};
const unsigned char DISC[] = {FLAG, ADDR_SX, CTRL_DISC, ADDR_SX ^
CTRL_DISC, FLAG};

static int open_port_called = FALSE;

// int last_was_set=0;

int llopen(LinkLayer connectionParameters)
{
    printf("llopen called\n");
    if (!open_port_called)
    {
        open_port_called = TRUE;
        if (openSerialPort(connectionParameters.serialPort,
                           connectionParameters.baudRate) < 0)
        {
            return -1;
        }

        // Set alarm function handler
        (void)signal(SIGALRM, alarmHandler);
    }
    MAX_ALARM_REPEATS = connectionParameters.nRetransmissions;
    TIMEOUT = connectionParameters.timeout;
    printf("Alarm set!\n");

    frame_num = 0;

    enum OPEN_STATE state = STATE_START;
    int run = TRUE;
    unsigned char buf, expected_address_flag, expected_code;

    if (connectionParameters.role == LlTx)
    {
        expected_address_flag = ADDR_SX;
        expected_code = CTRL_UA;
    }
    else
    {
        expected_address_flag = ADDR_SX;
        expected_code = CTRL_SET;
    }

    int has_received_bytes = 0;
```

```
    alarmCount = 0;
    while (run)
    {
        if (alarmEnabled == FALSE)
        {
            alarmEnabled = TRUE;
            if (connectionParameters.role == LlTx &&
!has_received_bytes)
            {
                writeBytesSerialPort(SET, SHORT_MESSAGE_SIZE);

                printf("Wrote set message!\n");
            }
            alarm(TIMEOUT);
        }
        if (alarmCount == MAX_ALARM_REPEATS)
        {
            run = FALSE;
        }
        int bytes = readByteSerialPort(&buf);
        if (bytes == 1)
        {
            has_received_bytes = 1;
            alarmCount = 0;
            switch (state)
            {
            case STATE_START:

                if (buf == FLAG)
                {
                    state = STATE_FLAG_RCV;

                    printf("read flag\n");
                }
                break;

            case STATE_FLAG_RCV:
                if (buf == FLAG)
                {
                    break;
                }
                if (buf == expected_address_flag)
                {
                    state = STATE_A_RCV;
                    printf("read address\n");
```

```
                    break;
                }
                state = STATE_START;
                break;

            case STATE_A_RCV:
                if (buf == FLAG)
                {
                    state = STATE_FLAG_RCV;
                    break;
                }
                if (buf == expected_code)
                {
                    state = STATE_C_RCV;
                    printf("read code\n");
                    break;
                }
                state = STATE_START;
                break;

            case STATE_C_RCV:
                if (buf == FLAG)
                {
                    state = STATE_FLAG_RCV;
                    break;
                }
                if (buf == (expected_address_flag ^ expected_code))
                {
                    state = STATE_BCC_OK;
                    printf("read error correction\n");
                    break;
                }
                state = STATE_START;
                break;

            case STATE_BCC_OK:
                if (buf == FLAG)
                {
                    printf("read final flag\n");
                    alarm(0);
                    alarmEnabled = FALSE;
                    if (connectionParameters.role == LlRx)
                    {
                        return writeBytesSerialPort(UA,
SHORT_MESSAGE_SIZE) > 0 ? 1 : -1;
                    }
```

```
                    else
                    {
                        return 1;
                    }
                }
                else
                {
                    state = STATE_START;
                }
                break;
            }
        }
        else if (bytes == -1)
        {
            return -1;
        }
    }

    return -1;
}

////////////////////////////////////////////////
// LLWRITE
////////////////////////////////////////////////

enum WRITE_STATE
{
    STATE_WRITE_START = 0,
    STATE_WRITE_FLAG_RCV = 1,
    STATE_WRITE_A_RCV = 2,
    STATE_WRITE_C_RCV = 3,
    STATE_WRITE_BCC_CORRECT = 4,
    STATE_WRITE_REPEAT_UA_RECEIVED = 5,
    STATE_WRITE_REPEAT_UA_BCC_CORRECT = 6
};

int llwrite(const unsigned char *buf, int bufSize)
{
    printf("frame ordering: %d\n", frame_num ? 1 : 0);
    alarmCount = 0;

    unsigned char to_send[bufSize + LLWRITE_EXTRA_BIT_NUM];
    to_send[0] = FLAG;
    to_send[1] = ADDR_SX;
    if (frame_num == 0)
    {
```

```
        to_send[2] = CTRL_I0;
    }
    else
    {
        to_send[2] = CTRL_I1;
    }
    to_send[3] = to_send[1] ^ to_send[2];
    int num_bytes = 4;

    unsigned char bcc2 = 0;
    for (size_t i = 0; i < bufSize; i++)
    {
        printf("byte: 0x%2x\n", buf[i]);
        bcc2 ^= buf[i];
        if (buf[i] == ESCAPE || buf[i] == FLAG)
        {
            to_send[num_bytes] = ESCAPE;
            num_bytes++;
            to_send[num_bytes] = buf[i] ^ SPECIAL_MASK;
            num_bytes++;
        }
        else
        {
            to_send[num_bytes] = buf[i];
            num_bytes++;
        }
    }

    printf("bcc2: %d (0x%2x)\n", bcc2, bcc2);
    if (bcc2 == ESCAPE || bcc2 == FLAG)
    {
        to_send[num_bytes] = ESCAPE;
        num_bytes++;
        to_send[num_bytes] = bcc2 ^ SPECIAL_MASK;
        printf("bcc2 transformed: %d\n", bcc2 ^ SPECIAL_MASK);
        num_bytes++;
    }
    else
    {
        to_send[num_bytes] = bcc2;
        num_bytes++;
    }
    to_send[num_bytes] = FLAG;
    num_bytes++;

    bytes_sent += num_bytes;
```

```
enum WRITE_STATE state = STATE_WRITE_START;
int run = TRUE;
unsigned char bt, code, num_tries = 0, rrLostTries = 0;

while (run)
{
    if (alarmEnabled == FALSE)
    {
        alarmEnabled = TRUE;
        swrite_calls++;
        if (writeBytesSerialPort(to_send, num_bytes) == -1)
        {
            return -1;
        }
        printf("sent message\n");
        actual_bytes_sent += num_bytes;
        alarm(TIMEOUT);
    }
    if (alarmCount >= MAX_ALARM_REPEATS)
    {
        run = FALSE;
    }

    int bytes = readByteSerialPort(&bt);
    if (bytes == 1)
    {
        alarmCount = 0;

        switch (state)
        {
        case STATE_WRITE_START:

            if (bt == FLAG)
            {
                state = STATE_WRITE_FLAG_RCV;
                printf("received flag (0x%2x)\n", bt);
                break;
            }
            printf("Should have been flag\n");
            break;

        case STATE_WRITE_FLAG_RCV:
            if (bt == FLAG)
            {
                printf("found flag instead of address\n");
```

```
                    break;
                }
                if (bt == ADDR_SX)
                {
                    state = STATE_WRITE_A_RCV;
                    printf("read address\n");
                    break;
                }
                printf("wrong address: 0x%2x\n", bt);
                state = STATE_WRITE_START;
                break;

            case STATE_WRITE_A_RCV:
                if (bt == FLAG)
                {
                    printf("found flag instead of command\n");
                    state = STATE_WRITE_FLAG_RCV;
                    break;
                }

                code = bt;
                if (code == CTRL_UA)
                {
                    state = STATE_WRITE_REPEAT_UA_RECEIVED;
                    printf("Random UA received!\n");
                    break;
                }

                if (frame_num == 0)
                {
                    if ((code == CTRL_REJ0) || (code == CTRL_RR1))
                    {
                        printf("Read correct command(0x%2x)\n", code);
                        state = STATE_WRITE_C_RCV;
                        break;
                    }
                    else if ((code == CTRL_REJ1) || (code == CTRL_RR0))
                    {
                        printf("Command read: out of sync!!\n");
                        if (num_tries < TRIES)
                        {
                            if (rrLostTries > RR_LOST_TRIES && code ==
CTRL_REJ1)
                            {
                                printf("Assumed rr lost, moving to next
frame\n");
```

```
                            frame_num = !frame_num;
                            return -2;
                        }
                        printf("Retrying rr reception\n");
                        state = STATE_WRITE_START;
                        rrLostTries++;
                        break;
                    }
                    else
                    {
                        printf("Skipping to next frame\n");
                        frame_num = !frame_num;
                        alarm(0);
                        alarmEnabled = FALSE;
                        return -2;
                    }
                }
                else
                {
                    printf("didn't read the correct command:
0x%2x\n", code);
                    if (rrLostTries < TRIES)
                    {
                        printf("Retrying rr reception\n");
                        state = STATE_WRITE_START;
                        rrLostTries++;
                        break;
                    }
                    else
                    {
                        printf("Serious error - exiting the
program\n\n");
                        frame_num = !frame_num;
                        alarm(0);
                        alarmEnabled = FALSE;
                        return -3;
                    }
                }
            }
            else
            {

                if ((code == CTRL_REJ1) || (code == CTRL_RR0))
                {
                    printf("Read correct command(0x%2x)\n", code);
                    state = STATE_WRITE_C_RCV;
```

```
                    break;
                }
                else if ((code == CTRL_REJ0) || (code == CTRL_RR1))
                {
                    printf("Command read: out of sync!!\n");
                    if (rrLostTries < TRIES)
                    {
                        if (rrLostTries > RR_LOST_TRIES && code ==
CTRL_REJ0)
                        {
                            printf("Assumed rr lost, moving to next
frame\n");
                            frame_num = !frame_num;
                            return -2;
                        }
                        printf("Retrying rr reception\n");
                        state = STATE_WRITE_START;
                        rrLostTries++;
                        break;
                    }
                    else
                    {
                        printf("Skipping to next frame\n");
                        frame_num = !frame_num;
                        alarm(0);
                        alarmEnabled = FALSE;
                        return -2;
                    }
                }
                else
                {

                    printf("didn't read the correct command:
0x%2x\n", code);
                    if (rrLostTries < TRIES)
                    {
                        printf("Retrying rr reception\n");
                        state = STATE_WRITE_START;
                        rrLostTries++;
                        break;
                    }
                    else
                    {
                        printf("Serious error - exiting the
program\n\n");
                        frame_num = !frame_num;
```

```
                            alarm(0);
                            alarmEnabled = FALSE;
                            return -3;
                        }
                    }
                }

                state = STATE_WRITE_START;
                break;

            case STATE_WRITE_C_RCV:
                if (bt == FLAG)
                {
                    printf("found flag instead of bcc\n");
                    state = STATE_WRITE_FLAG_RCV;
                    break;
                }
                if (bt == (ADDR_SX ^ code))
                {
                    state = STATE_WRITE_BCC_CORRECT;
                    printf("read correct bcc\n");
                    break;
                }
                printf("problem in error correction: %d read, %d
expected\n", bt, ADDR_SX ^ code);
                state = STATE_WRITE_START;
                break;

            case STATE_WRITE_REPEAT_UA_RECEIVED:
                if (bt == FLAG)
                {
                    state = STATE_WRITE_FLAG_RCV;
                    break;
                }
                if (bt == (ADDR_SX ^ CTRL_UA))
                {
                    state = STATE_WRITE_BCC_CORRECT;
                    printf("read error correction (repeat ua)\n");
                    break;
                }
                state = STATE_WRITE_START;
                break;
            case STATE_WRITE_REPEAT_UA_BCC_CORRECT:
                state = STATE_WRITE_START;
                break;
            case STATE_WRITE_BCC_CORRECT:
```

```
                if (bt == FLAG)
                {
                    printf("read final flag\n");

                    if (frame_num == 0)
                    {
                        if (code == CTRL_REJ0)
                        {
                            printf("resend 0\n");
                            errors_read += 1;
                            state = STATE_WRITE_START;
                            alarmCount = 0;
                        }
                        else if (code == CTRL_RR1)
                        {

                            printf("send next 1\n\n");
                            frame_num = !frame_num;
                            alarm(0);
                            alarmEnabled = FALSE;

                            return num_bytes;
                        }
                        printf("shouldn't happen without resend 0\n");
                        // WHAT HAPPENS IF RRO?
                    }
                    else
                    {
                        if (code == CTRL_REJ1)
                        {
                            printf("resend 1\n");
                            errors_read += 1;
                            state = STATE_WRITE_START;
                            alarmCount = 0;
                        }
                        else if (code == CTRL_RR0)
                        {

                            frame_num = !frame_num;
                            alarm(0);
                            alarmEnabled = FALSE;
                            printf("send next 0\n\n");

                            return num_bytes;
                        }
                        printf("shouldn't happen without resend 1\n");
```

```
                        // WHAT HAPPENS IF RR1?
                    }
                }
                else
                {
                    printf("Didn't read final flag - resetting\n");
                    state = STATE_WRITE_START;
                }
                break;
            }
        }
        else if (bytes == -1)
        {
            return -1;
        }
    }
    printf("write timeout\n");
    return -1;
}

////////////////////////////////////////////////
// LLREAD
////////////////////////////////////////////////
enum RTERM_STATE
{
    RTERM_STATE_START = 0,
    RTERM_STATE_FLAG_RCV = 1,
    RTERM_STATE_A_RCV = 2,
    RTERM_STATE_C_RCV = 3,
    RTERM_STATE_BCC_OK = 4,
};

int terminate_reader()
{

    if (writeBytesSerialPort(DISC, SHORT_MESSAGE_SIZE) == -1)
    {

        return -1;
    }

    frame_num = 0;

    enum RTERM_STATE state = RTERM_STATE_START;
    int run = TRUE;
    unsigned char buf, expected_address_flag = ADDR_SX, expected_code =
```

```
CTRL_UA;

    while (run)
    {
        if (alarmEnabled == FALSE)
        {
            alarmEnabled = TRUE;
            alarm(TIMEOUT);
        }
        if (alarmCount == MAX_ALARM_REPEATS)
        {
            run = FALSE;
        }
        int bytes = readByteSerialPort(&buf);
        if (bytes == 1)
        {
            alarmCount = 0;
            switch (state)
            {
            case RTERM_STATE_START:
                if (buf == FLAG)
                {
                    state = RTERM_STATE_FLAG_RCV;

                    printf("term read flag\n");
                    break;
                }
                printf("Should have been flag\n");
                break;

            case RTERM_STATE_FLAG_RCV:
                if (buf == FLAG)
                {
                    break;
                }
                if (buf == expected_address_flag)
                {
                    state = RTERM_STATE_A_RCV;
                    printf("term read address\n");
                    break;
                }
                state = RTERM_STATE_START;
                break;

            case RTERM_STATE_A_RCV:
                if (buf == FLAG)
```

```
            {
                state = RTERM_STATE_FLAG_RCV;
                break;
            }
            if (buf == expected_code)
            {
                state = RTERM_STATE_C_RCV;
                printf("term read code\n");
                break;
            }
            state = RTERM_STATE_START;
            break;

        case RTERM_STATE_C_RCV:
            if (buf == FLAG)
            {
                state = RTERM_STATE_FLAG_RCV;
                break;
            }
            if (buf == (expected_address_flag ^ expected_code))
            {
                state = RTERM_STATE_BCC_OK;
                printf("term read error correction\n");
                break;
            }
            state = RTERM_STATE_START;
            break;

        case RTERM_STATE_BCC_OK:
            if (buf == FLAG)
            {
                printf("term read final flag\n");
                ;
                alarmEnabled = FALSE;
                alarmCount = 0;
                open_port_called = FALSE;
                closeSerialPort();
                return 1;
            }
            else
            {
                state = RTERM_STATE_START;
            }
            break;
        }
    }
```

```
        else if (bytes == -1)
        {
            return -1;
        }
    }

    return -1;
}

enum READ_STATE
{
    STATE_READ_START = 0,
    STATE_READ_FLAG_RCV = 1,
    STATE_READ_A_RCV = 2,
    STATE_READ_C_RCV = 3,
    STATE_READ_DATA = 4,
    STATE_READ_ESCAPED = 5,
    STATE_READ_DISC = 6,
    STATE_READ_DISC_BCC_OK = 7,
    STATE_READ_SET = 8,
    STATE_READ_SET_BCC_OK = 9
};

int data_is_correct(unsigned char *data, unsigned int data_length,
unsigned char bcc2)
{
    unsigned char res = 0;
    for (size_t i = 0; i < data_length; i++)
    {
        res ^= data[i];
    }
    printf("bcc2 created: %d, bcc2 received with message:%d\n", res,
bcc2);
    return res == bcc2;
}

int llread(unsigned char *packet) // buffer already instantiated
{
    printf("frame ordering: %d\n", frame_num ? 1 : 0);

    alarmCount = 0;
    enum READ_STATE state = 0;
    int run = TRUE;

    unsigned char buf, expected_address_flag = ADDR_SX, expected_code,
expected_rej, expected_rr, out_of_order_frame_code, received_code,
```

```
attemptCount = 0;
    unsigned int current_data_index = 0;

    if (frame_num == 0)
    {
        expected_code = CTRL_I0;
        out_of_order_frame_code = CTRL_I1;
        expected_rej = CTRL_REJ0;
        expected_rr = CTRL_RR1;
        received_code = expected_code;
    }
    else
    {
        expected_code = CTRL_I1;
        received_code = expected_code;
        out_of_order_frame_code = CTRL_I0;
        expected_rej = CTRL_REJ1;
        expected_rr = CTRL_RR0;
    }

    while (run)
    {
        if (alarmEnabled == FALSE)
        {
            alarmEnabled = TRUE;
            alarm(TIMEOUT);
        }
        if (alarmCount >= MAX_ALARM_REPEATS)
        {
            run = FALSE;
        }
        int bytes = readByteSerialPort(&buf);
        if (bytes == 1)
        {
            alarmCount = 0;

            if (current_data_index > MAX_PAYLOAD_SIZE)
            {
                printf("Overflow danger: end flag not found for too
long!!!\n");
                return -1;
            }
            switch (state)
            {
            case STATE_READ_START:
```

```
            if (buf == FLAG)
            {
                state = STATE_READ_FLAG_RCV;

                printf("read flag 1\n");
            }
            break;

        case STATE_READ_FLAG_RCV:
            if (buf == FLAG)
            {
                printf("found flag instead of address\n");
                break;
            }
            if (buf == expected_address_flag)
            {
                state = STATE_READ_A_RCV;
                printf("read address\n");
                break;
            }
            printf("read wrong address:  0x%2x\n", buf);
            state = STATE_READ_START;
            break;

        case STATE_READ_A_RCV:
            if (buf == FLAG)
            {
                state = STATE_READ_FLAG_RCV;
                printf("found flag instead of code\n");
                break;
            }
            if (buf == CTRL_SET)
            {
                received_code = CTRL_SET;
                state = STATE_READ_SET;
                printf("read set code\n");
                break;
            }
            if (buf == CTRL_DISC)
            {
                received_code = CTRL_DISC;
                state = STATE_READ_DISC;
                printf("read disc code\n");
                break;
            }
            if (buf == expected_code)
```

```
                {
                    received_code = expected_code;
                    state = STATE_READ_C_RCV;
                    printf("read frame code\n");
                    break;
                }
                if (buf == out_of_order_frame_code)
                {
                    received_code = out_of_order_frame_code;
                    state = STATE_READ_C_RCV;
                    printf("read out of order frame code\n");
                    break;
                }
                printf("Read wrong code: 0x%2x\n", buf);
                received_code = buf;
                state = STATE_READ_C_RCV; // TODO: I don't like this,
but it has to be, just in case it's a data frame. I don't want
"arbitrary code execution" here at all
                break;
            case STATE_READ_SET:

                if (buf == FLAG)
                {

                    printf("read flag instead of bcc!!!\n");

                    state = STATE_READ_FLAG_RCV;
                    break;
                }
                if (buf == (expected_address_flag ^ received_code))
                {
                    state = STATE_READ_SET_BCC_OK;
                    printf("read set bbc ok\n");
                    break;
                }
                printf("(set) bcc wrong\n");
                state = STATE_READ_DATA;
                break;
            case STATE_READ_SET_BCC_OK:
                if (buf == FLAG)
                {
                    frame_num = 0; //?Because Reset?
                    alarm(0);
                    alarmEnabled = FALSE;
                    alarmCount = 0;
                    writeBytesSerialPort(UA, SHORT_MESSAGE_SIZE);
```

```
                    printf("Had to send another UA!");
                    return -4; // also not a documented return value,
but could be useful
                }
                printf("Didn't find the final flag of a set command!");

                state = STATE_READ_DATA; // TODO: I hate that I have to
do this, but I have no alternative. Many errors could happen if I didn't
                break;

            case STATE_READ_DISC:

                if (buf == FLAG)
                {
                    state = STATE_READ_FLAG_RCV;
                    break;
                }
                if (buf == (expected_address_flag ^ received_code))
                {
                    state = STATE_READ_DISC_BCC_OK;
                    printf("bbc ok\n");
                    break;
                }
                state = STATE_READ_START;
                break;

            case STATE_READ_DISC_BCC_OK:
                if (buf == FLAG)
                {
                    printf("terminating reader function called!\n");
                    return (terminate_reader() == 1) ? -2 : -3;
                }
                state = STATE_READ_START;

                break;

            case STATE_READ_C_RCV:
                if (buf == FLAG)
                {
                    printf("Found flag instead of bcc\n");
                    state = STATE_READ_FLAG_RCV;
                    break;
                }
                if (buf == (expected_address_flag ^ received_code))
                {
```

```
                    state = STATE_READ_DATA;

                    printf("bcc correct\n");
                    break;
                }
                printf("bcc incorrect\n");
                if (received_code != CTRL_DISC && received_code !=
CTRL_SET)
                {
                    printf("Assuming data frame -> must not be induced
in error due to possible frame content\n");
                    state = STATE_READ_DATA; // will this fix the
problems?
                    break;
                }
                state = STATE_READ_START;
                break;

            case STATE_READ_DATA:
                if (buf == FLAG)
                {
                    printf("read final flag\n");

                    if (received_code == out_of_order_frame_code)
                    {
                        printf("out of order frame received\n");
                        current_data_index = 0;
                        int rs;
                        if (expected_rej == CTRL_REJ0)
                        {
                            printf("rej0\n");
                            rs = writeBytesSerialPort(REJ0,
SHORT_MESSAGE_SIZE);
                        }
                        else
                        {
                            printf("rej1\n");
                            rs = writeBytesSerialPort(REJ1,
SHORT_MESSAGE_SIZE);
                        }
                        // TODO: maybe don't send this?
                        if (rs == -1)
                        {
                            printf("Error sending REJ\n");
                            if (attemptCount < TRIES)
                            {
```

```
                                attemptCount++;
                                printf("Retrying reception\n");
                                state = STATE_READ_START;
                                break;
                            }
                            else
                            {
                                alarm(0);
                                alarmEnabled = FALSE;

                                printf("Irrecoverable send REJ
error\n\n");

                                return -1;
                            }
                        }

                        if (attemptCount < TRIES)
                        {
                            attemptCount++;
                            printf("Retrying reception\n");
                            state = STATE_READ_START;
                            break;
                        }
                        else
                        {
                            alarm(0);
                            alarmEnabled = FALSE;

                            printf("Irrecoverable sync error\n\n");

                            return -1;
                        }
                    }

                    if (received_code != expected_code)
                    {
                        current_data_index = 0;

                        if (attemptCount < TRIES)
                        {
                            attemptCount++;
                            printf("Retrying reception due to error
control\n");
                            state = STATE_READ_START;
                            break;
```

```
                        }
                        else
                        {
                            alarm(0);
                            alarmEnabled = FALSE;

                            printf("Irrecoverable command related
error\n\n");

                            return -1;
                        }
                    }

                    current_data_index--;
                    if (data_is_correct(packet, current_data_index,
packet[current_data_index]))
                    {
                        printf("data received!\n");
                        int res;
                        if (expected_rr == CTRL_RR0)
                        {
                            res = writeBytesSerialPort(RR0,
SHORT_MESSAGE_SIZE);

                            printf("sent rr0\n");
                        }
                        else
                        {
                            res = writeBytesSerialPort(RR1,
SHORT_MESSAGE_SIZE);
                            printf("sent rr1\n");
                        }
                        if (res != -1)
                        {
                            alarm(0);
                            alarmEnabled = FALSE;
                            frame_num = !frame_num;
                            return current_data_index;
                        }
                        alarm(0);
                        alarmEnabled = FALSE;
                        printf("error in sending rr");
                        return -1;
                    }
                    else
                    {
```

```
                        printf("bbc2 incorrect!\n");
                        if (expected_rej == CTRL_REJ0)
                        {
                            writeBytesSerialPort(REJ0,
SHORT_MESSAGE_SIZE);
                        }
                        else
                        {
                            writeBytesSerialPort(REJ1,
SHORT_MESSAGE_SIZE);
                        }

                        if (attemptCount < TRIES)
                        {
                            attemptCount++;
                            current_data_index = 0;
                            state = STATE_READ_START;
                            printf("--> Retrying reception: %d/%d\n",
attemptCount, TRIES);
                            break;
                        }
                        else
                        {
                            attemptCount += 1;
                            alarm(0);
                            alarmEnabled = FALSE;

                            return -1;
                        }
                    }
                }
                else if (buf == ESCAPE)
                {
                    printf("escaped\n");
                    state = STATE_READ_ESCAPED;
                }
                else
                {
                    packet[current_data_index] = buf;
                    current_data_index++;
                    printf("byte: 0x%2x\n", packet[current_data_index -
1]);
                }
                break;

            case STATE_READ_ESCAPED:
```

```
                packet[current_data_index] = buf ^ SPECIAL_MASK;
                printf("byte: 0x%2x\n", packet[current_data_index]);
                current_data_index++;
                state = STATE_READ_DATA;
                break;
            default:
                printf("I quit,\n");
                return -1;
                break;
            }
        }
        else if (bytes == -1)
        {
            return -1;
        }
    }

    return -1;
}

////////////////////////////////////////////////
// LLCLOSE
////////////////////////////////////////////////
enum CLOSE_STATE
{
    CLOSE_STATE_START = 0,
    CLOSE_STATE_FLAG_RCV = 1,
    CLOSE_STATE_A_RCV = 2,
    CLOSE_STATE_C_RCV = 3,
    CLOSE_STATE_BCC_OK = 4,
};
int llclose(int showStatistics)
{

    if (showStatistics == TRUE)
    {
        printf("Wrote %u unique bytes(%u counting repeated sends), with
%u calls to llwrite (repeated frames included).\n", bytes_sent,
               actual_bytes_sent, swrite_calls);
    }
    enum CLOSE_STATE state = CLOSE_STATE_START;
    int run = TRUE;

    unsigned char buf, expected_address_flag = ADDR_SX, expected_code =
CTRL_DISC;
    alarmCount = 0;
```

```
    while (run)
    {
        if (alarmEnabled == FALSE)
        {
            printf("wrote disc\n");
            alarmEnabled = TRUE;
            if (writeBytesSerialPort(DISC, SHORT_MESSAGE_SIZE) == -1)
            {

                return -1;
            }
            alarm(TIMEOUT);
        }
        if (alarmCount == MAX_ALARM_REPEATS)
        {
            run = FALSE;
        }
        int bytes = readByteSerialPort(&buf);
        if (bytes == 1)
        {

            alarmCount = 0;
            switch (state)
            {
            case CLOSE_STATE_START:

                if (buf == FLAG)
                {
                    state = CLOSE_STATE_FLAG_RCV;

                    printf("read flag\n");
                }
                break;

            case CLOSE_STATE_FLAG_RCV:
                if (buf == FLAG)
                {
                    break;
                }
                if (buf == expected_address_flag)
                {
                    state = CLOSE_STATE_A_RCV;
                    printf("read address\n");
                    break;
                }
                state = CLOSE_STATE_START;
```

```
                break;

            case CLOSE_STATE_A_RCV:
                if (buf == FLAG)
                {
                    state = CLOSE_STATE_FLAG_RCV;
                    break;
                }
                if (buf == expected_code)
                {
                    state = CLOSE_STATE_C_RCV;
                    printf("read code\n");
                    break;
                }
                state = CLOSE_STATE_START;
                break;

            case CLOSE_STATE_C_RCV:
                if (buf == FLAG)
                {
                    state = CLOSE_STATE_FLAG_RCV;
                    break;
                }
                if (buf == (expected_address_flag ^ expected_code))
                {
                    state = CLOSE_STATE_BCC_OK;
                    printf("read error correction\n");
                    break;
                }
                state = CLOSE_STATE_START;
                break;

            case CLOSE_STATE_BCC_OK:
                if (buf == FLAG)
                {
                    printf("read final flag\n");

                    alarmEnabled = FALSE;
                    alarmCount = 0;
                    writeBytesSerialPort(UA, SHORT_MESSAGE_SIZE);
                    writeBytesSerialPort(UA, SHORT_MESSAGE_SIZE); //
just in case the first isn't read, so that the receive doesn't terminate
with errors :/
                    writeBytesSerialPort(UA, SHORT_MESSAGE_SIZE);
                    run = 0;
                    open_port_called = FALSE;
```

```
                    int clstat = closeSerialPort();
                    return clstat != -1 ? 1 : -1;
                }
                else
                {
                    state = CLOSE_STATE_START;
                }
                break;
            }
        }
        else if (bytes == -1)
        {
            return -1;
        }
    }

    return -1;
}
```

Appendix II - Test Scripts

- test_performance.sh

```
#!/bin/bash
./test_performance_rx.sh & ./test_performance_tx.sh
```

- test_performance_rx.sh

```
#!/bin/bash
echo "result,exec_time" > performance.rx
for n in {1..10};
do
CODE_TIME="$(time (make run_rx) 2>&1)"
CODE_RESULT=$?
echo "$CODE_RESULT,${CODE_TIME:(-44)}" >> performance.rx
sleep 1
done
```

-

- test_performance_tx.sh

```
#!/bin/bash
echo "result,exec_time" > performance.tx
for n in {1..10};
do
CODE_TIME="$(time (make run_tx) 2>&1)"
CODE_RESULT=$?
echo "$CODE_RESULT,${CODE_TIME:(-44)}" >> performance.tx
sleep 1
done
```