Faculdade de Engenharia da Universidade do Porto

Lab 1 - Data Link Protocol and Application for transferring files

Redes de Computadores L.EIC025-2024/2025-1S:

Regente: Manuel Alberto Pereira Ricardo

Regente: Rui Pedro de Magalhães Claro Prior

Class 7:

Professor: Filipe Miguel Monteiro Da Silva E Sousa

Authors:

David dos Santos Carvalho Ranito up202206312@fe.up.pt

Pedro Daniel Freitas João up202204962@fe.up.pt

Tiago Freitas Torres up202208938@fe.up.pt

Delivery Date:

10/11/2024

Index

Summary

This project was realized for the Curricular Unit of Computer Networks which aims for the implementation of a communication protocol for the transmission of files through a RS-232 Serial Port cable.

Throughout the development of this project, we applied and reinforced the contents covered in the theoretical classes including protocol design, error detection and correction, data link layer mechanisms, and reliable data transfer techniques.

Introduction

The objective of this project is to implement a data link layer and the application layer for transferring a file stored in one computer to another through a RS-232 serial cable.

It is also important that the data link layer provides a reliable communication between both systems by implementing the Stop & Wait protocol for error handling.

In this report we start by explaining the code structure and main functional blocks, before we describe some common use cases and their function call sequence. Then we explain in more detail how we implemented the logical link and application protocols and their functionalities. We finish the report with a brief description of the tests performed on our protocols and the data link protocol efficiency.

Architecture and Code Structure

There are 2 primary functional blocks: the data link layer and the application layer, and 2 interfaces: the data link layer interface and the serial port API, although the latter was provided and not implemented by us.

We will present them in a lower level to higher level order.

Serial Port API

The serial port API *(serial_port.h)* that was provided, is used to interact with the serial port. This includes opening and configuring the serial port (*openSerialPort()*), writing bytes (*writeBytes()),* reading a byte (*readByte()*) and closing the serial port (*closeSerialPort()).*

Data Link Layer

The data link layer *(link_layer.c*) implements transmitter and receiver functionality for transferring data and uses the serial port API for the serial port communication.

We store the statistics of the current connection in a struct *comms_stats* that contains the number of frames sent, frames received with and without errors, frames received with error, number of retransmissions, timeouts, duplicate data frames received, data frames sent, data frames received with or without errors. The last 3 statistics are only available for the receiver and the number of data frames sent is only available for the transmitter.

Data Link Layer Interface

The data link layer interface (*link_layer.h*) provides the necessary functions for establishing and using a reliable connection between transmitter and receiver.

To store the connection parameters the struct *LinkLayer* is used that contains the serial port identification, the role in the transfer, the baud rate, the number of retransmissions in case of failure and a timeout value, in seconds, to wait until retransmission, after not receiving response.

These functions are *llopen()* to open a connection according to the parameters sent in the LinkLayer struct, *llwrite*() to to send data, *llread*() to read data and *llclose*() to close the previously opened connection and optionally print the statistics.

Application Layer

The application layer is responsible for a higher-level logic that reads/writes a file, which will be divided into fragments. Each fragment will be sent from the transmitter to the receiver, using the functionality provided by the data link layer.

Additional data about the file is sent by control packets, and, in data packets, each fragment is preceded by information that helps keep file integrity.

The *applicationLayer*() function handles both the receiving and sending cases, and either builds or parses the packets.

Main Use Cases

The two main use cases are the transfer and the receival of a file. The program flow differs between them.

Transmitter:

1. *llopen*() : Called by *applicationLayer*(), passes Tx as role and is responsible for establishing the connection between receiver and transmitter.
   a. *sendSupervision*() : Sends the supervision frame that initiates the connection
   b. *receiveSupervision*(): Receives the supervision frame that confirms the connection has been established
2. *llwrite*() : Sends the packet to the data link layer. Will be called for the start packet, all the data packets and the end packet.
   a. *prepareFrame*(): Adds frame headers and does byte stuffing.
   b. *writeBytes*(): Sends the bytes to the serial port.
   c. *waitWriteResponse*(): Waits for the receiver response using a state machine.
3. *llclose*() : Closes the connection between receiver and transmitter.
   a. *sendSupervision*() : Sends a DISC frame to signal the receiver that the connection should end.
   b. *receiveSupervision*() : Waits for a DISC frame sent by the receiver.
   c. *sendSupervision*() : Finally sends an UA frame to confirm the end of connection.

Receiver:

1. *llopen*() : Called by *applicationLayer*(), passes Rx as role and is responsible for establishing the connection between receiver and transmitter.

2. *llread*() : Data link layer reads a packet using a state machine that handles communication errors. Used to read all control and data packets.
   a. *readByte*() : Reads a byte from serial port, called for every byte received.
   b. *sendSupervision*() : Depending on the data read, if there is an error on the data read sends RJ, otherwise sends REJ frame.
3. *llcose*() : Closes the connection between receiver and transmitter.
   a. *receiveSupervision*() : Waits for the DISC frame sent by transmitter.
   b. *sendSupervision*() : Sends DISC frame.
   c. *waitDiscResponse*() : Waits for final UA frame.

Logical Link Protocol

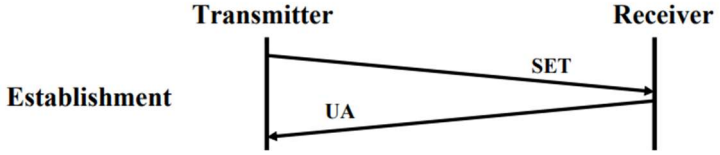
Open Connection

Establishing the connection between the transmitter and the receive, is done through function *llopen()*.

Initially the opening and configuration of the serial port, as specified in the *LinkLayer* struct, is performed by calling *openSerialPort().* The transmitter initiates the connection by sending SET supervision frame, and waiting for the UA supervision frame, which is an acknowledgment reply from the receiver.

When a supervision frame is sent, an alarm is enabled to manage timeouts and retransmissions. If the acknowledgment frame is received within the timeout period, the alarm is cleared, confirming a successful connection. If the acknowledgment is not received within the timeout, the alarm triggers a retry by retransmitting the SET frame. This process continues until the maximum number of retransmissions is reached.

On the other hand, the receiver waits for the SET frame and upon receiving it, the receiver responds by sending back a UA frame, confirming the connection. This frame exchange establishes a reliable connection between the two endpoints, enabling further data communication



Write Data

The Information frame was assembled by calling prepare_frame() that also did the byte stuffing. When reading the data to send if the byte was the flag or the escape octet then we put an escape octet and the exclusive or of the byte read and 0x20 in the buffer. The stuffing was also done one last time for the bcc2 that was calculated from the data bytes before stuffed.

When the frame is assembled we send it and wait for a response by calling the *waitWriteResponse()* function and entering in a state machine that only stops when there is a timeout (returns 2) or when a REJ (returns 1) or RR (returns 0) are received.

If there's a timeout there are two possibilities, if the number of retransmissions is the previously established maximum then we exit *llwrite*() and return -1 else we retransmit the frame.

When receiving either a REJ or a RR the number of retransmissions goes back to 0 and in the case of the REJ the frame is retransmitted, in the case of a RR we update the information number to be sent and exit the function successfully.

Read Data

The whole process of reading an Information Frame, is done through calling the function *llread*() that builds the array of character read, packet, and returns the number of characters read, if successful.

The process operates as a state machine, where each state represents the parsing of frame field. In each state, a verification is done to ensure the correctness of the fields before proceeding to the next state, in case of failing the verification we retrieve to one of the initial states.

The control field indicates the type of frame, if the frame received is equal to the expected, it continues to the next state, but if it isn't an acknowledgement byte is sent, either RR1 or RR0, indicating that the receiver is ready to receive an information frame either number 0 or 1.

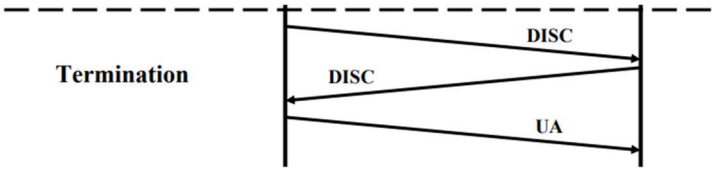
Next, the BCC verification is done, checking its equality to an XOR between address field and control field.

Following, we start reading the data packet generated by the application. For the first part, we check if the current byte is an escape octet, which means the next byte requires "destuffing" and inserted into the packet array. This process continues until the flag byte is reached. For the second part, we verify if the BCC from data (bcc2), is equal to the previous one received, if it is a reply is sent indicating to the transmitter the availability to receive the next information frame with the "opposite" type and reading of the frame ends. However, if it is not equal, the receiver replies by sending an indication of rejection (REJ) of the current frame, 0 or 1, returning to the initial state.

```
if (byte == FLAG) {
    //verificar bbc2, criar funcao
    int i = char_read - 1;
    unsigned char bcc2 = packet[i--];

    while (i>=0){
        bcc ^= packet[i--];

    }
    if (bcc == bcc2){

        frame_expected ^= 0x01; //want to receive
        sendSupervision(A_TX, RR(frame_expected));

        state = STOP_RCV;
    }
    else{
        if(frame_expected) {
            sendSupervision(A_TX, REJ1);
        }

    else {
            sendSupervision (A_TX,REJ0);
        }
        char_read = 0;
        bcc = 0;
        state = START_RCV;
    }

} else if (byte == ESC) {
    escape_next = 1;
} else {
    if (escape_next) {
        byte ^= 0x20;
        escape_next = 0;
    }
    packet[char_read++] = byte;
}
```

Close Connection

The connection is closed with this sequence of frames.



When closing the transmission the supervision frames are sent by calling *sendSupervision()* that receives the address and the control field. The *receiveSupervision()* is used for receiving a supervision frame with the address and control field given as arguments. This function also has an additional boolean argument

(timeout) that when false only exits when the frame is received and when true it can also exit when a timeout happens.

When closing as the transmitter a DISC frame is sent, the alarm is enabled and a call is made to *receiveSupervision()* (with timeout TRUE). If DISC is received then a UA frame is sent and *closeSerialPort()* is called to close the serial port.

When closing as the receiver *receiveSupervision*() (with timeout FALSE) is called to wait for DISC. After receiving DISC, it sends a DISC frame, the alarm is enabled and calls *waitDiscResponse()*. This function is similar to *waitWriteResponse(),* returns 2 if a timeout occurs, 1 if DISC is received and 0 if UA is received. We implemented the timeout mechanism here because in case a UA frame from the transmitter gets lost and because the transmitter exits after sending it the receiver would keep waiting for response forever and the program would not end.

If there's a timeout there are two possibilities, if the number of retransmissions is the previously established maximum then we call *closeSerialPort()* and return -1 else we retransmit the DISC.

If DISC is received then we retransmit the DISC and if UA is received we call *closeSerialPort()* to close the serial port.

Before returning the connection, statistics are printed if this option was activated.

Application Protocol

Open Connection

To start the connection, the *llopen()* function provided by the data link layer is used, and the configuration that the user chose is passed to it by a *LinkLayer* structure.

```
LinkLayer connectionParameters = {
    .baudRate = baudRate,
    .nRetransmissions = nTries,
    .timeout = timeout
};
strncpy(connectionParameters.serialPort,
        serialPort,
        sizeof(connectionParameters.serialPort) - 1);
```

Write Data

There are 3 main steps to transmitting a file: the start packet, the data packets and the end packet.

- Start packet:

Contains information about the file to be transmitted, such as size and filename. The first byte set to 1 indicates that it is a start packet.

First, the file information must be queried, and then the packet can be built. We take use of the *stat()* syscall to get the file size.

```
struct stat st;
stat(filename, &st);
int file_sz = st.st_size;
int filename_sz = strlen(filename);
```

The start packet also contains the size of the "size" field and the "filename" field. We chose to represent the "size" field by an int, so 4 bytes.

With all the data needed, the start packet is built and sent with *llwrite()*.

```
// START PACKET ASSEMBLY
unsigned char ctrl_pckt[256] = {0};
ctrl_pckt[0] = PCKT_C_START;
ctrl_pckt[1] = PCKT_T_FILE_SZ;
ctrl_pckt[2] = sizeof(int);
memcpy(&ctrl_pckt[3], &file_sz, sizeof(int));
ctrl_pckt[7] = PCKT_T_FILE_NM;
ctrl_pckt[8] = filename_sz;
memcpy(&ctrl_pckt[9], filename, filename_sz);

if (!llwrite(ctrl_pckt, 5 + sizeof(int) + filename_sz)){
    return;
}
```

- Data packets

The file will be read fragment by fragment, and each one will be sent in a data packet. Each packet will have a sequence byte from 0 to 99 that will be used by the receiver to check the file integrity. The data packet also contains the number of bytes per fragment, in our case 512, indicated in the L1 and L2 fields. The first byte of a data packet is 2.

```
#define FRAGMENT_SZ 512
#define DATA_PCKT_SZ (FRAGMENT_SZ + 4)
#define PCKT_L2 (FRAGMENT_SZ / 256)
#define PCKT_L1 (FRAGMENT_SZ % 256)
```

The number of packets that will be sent depend on the size of the file and the fragment size, and can be calculated. After that, a loop is run and each fragment is read from the file and sent with *llwrite()*

```
// DATA PACKETS ASSEMBLY
int n_fragments = (file_sz + FRAGMENT_SZ - 1) / FRAGMENT_SZ;
unsigned char data_pckt[DATA_PCKT_SZ] = {0};
int n = 0;
while (n < n_fragments){
    memset(data_pckt, 0, DATA_PCKT_SZ);
    data_pckt[0] = PCKT_C_DATA;
    int fragment_sz = read(fd_data, &data_pckt[4], FRAGMENT_SZ);

    data_pckt[1] = n % 100;
    data_pckt[2] = fragment_sz / 256;
    data_pckt[3] = fragment_sz % 256;

    n++;
    llwrite(data_pckt,4+fragment_sz);
```

- End packet

The end packet will be similar to the start packet, but the first byte will be a 3 instead of a one. After successfully sending it with *llwrite(),* the file is assumed to be fully transmitted.

Read Data

To read a file, all 3 types of packets must be received and parsed.

- Start packet

It will check if the first byte is 1, then it checks if the other fields represent the file size and filename and saves those values. The filename received in the start packet can also be chosen to name the file to be written.

```
// Filename given by transmitter or given by argument in receiver?
//int fd_target = open(filename_tx, O_WRONLY | O_CREAT, S_IRUSR);
int fd_target = open(filename, O_WRONLY | O_CREAT, 0666);
```

- Data packets

The program loops and receives packets with *llread()* until looking for data packets. If the sequence byte is correct, the received bytes are appended to the file. The L1 and L2 bytes are used to know how many bytes to write.

```
// DATA PACKETS RECEIVE
while (TRUE){
    llread(data_pckt);

    if ((data_pckt[0] == PCKT_C_DATA)
            &&
            (data_pckt[1] == (seq_n + 1)%100)){
        seq_n = data_pckt[1];
        l2 = data_pckt[2];
        l1 = data_pckt[3];
        bytes_read += write(fd_target,
                            &data_pckt[4],
                            l2*256+l1);
    }
    // END PACKET RECEIVE
    else if (data_pckt[0] == PCKT_C_END)
    {
        if (bytes_read != file_sz)
            printf("ERROR: File is not complete\n");
        break;
    }

}
```

- End packet

If an end packet is detected, the file size received in the start packet is checked against the number of bytes written so far. If they are the same, the file transfer is assumed to be over.

Close Connection

To close the connection, the *llclose()* data link layer is called.

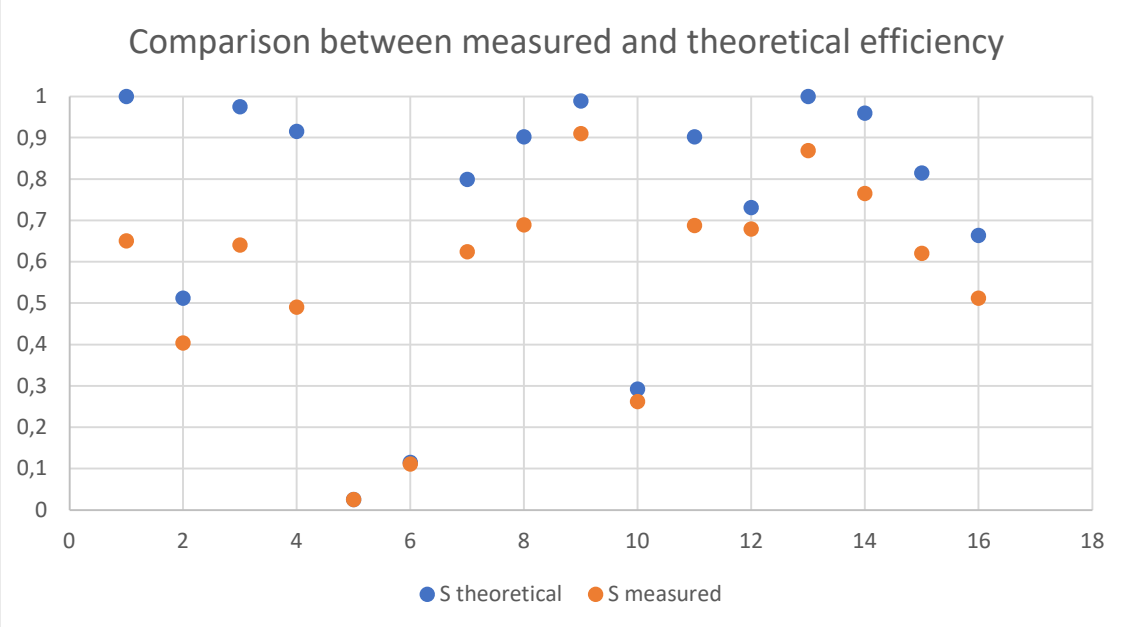
Data Link Protocol Efficiency and Validation

Here are some definitions that we used for measuring efficiency:

- L – size of frame in bytes
- BER – bit error rate
- FER – frame error rate, calculated as 1 - (1 - BER)^(8 * L)
- C – capacity of the link in bit/sec, in our case equal to the baud rate
- Tp – propagation time in microsec
- T measured – transference time measured in sec
- R – received bitrate in bits/sec
- S measured – measured efficiency, calculated as R / C
- S theoretical – theoretical efficiency, calculated as shown in classes

We varied L, BER, C and Tp and measured the time it took to transfer the file. In the following table each line is one measure with those statistics, and in the next graphic we can view in a clearer way the comparison between the measured efficiency and the theoretical efficiency for each combination of statistics.

L (bytes)	BER	FER	C (bits/sec)	Tp (microsec)	T measured (sec)	R (bits/sec)	S measured	S theoretical
64	0	**0,000**	**1200**	**0**	**112,30**	781	**0,651**	**1,000**
64	0,00001	**0,005**	**2400**	**100000**	**90,49**	970	**0,404**	**0,513**
64	0,00005	**0,025**	**9600**	**0**	**14,25**	6157	**0,641**	**0,975**
64	0,0001	**0,050**	**9600**	**1000**	**18,62**	4713	**0,491**	**0,916**
128	0	**0,000**	**19200**	**1000000**	**178,74**	491	**0,026**	**0,026**
128	0,00001	**0,010**	**38400**	**100000**	**20,44**	4293	**0,112**	**0,116**
128	0,00005	**0,050**	**9600**	**10000**	**14,62**	6001	**0,625**	**0,800**
128	0,0001	**0,097**	**9600**	**0**	**13,24**	6626	**0,690**	**0,903**
256	0	**0,000**	**115200**	**100**	**0,84**	104844	**0,910**	**0,989**
256	0,00001	**0,020**	**2400**	**1000000**	**139,06**	631	**0,263**	**0,293**
256	0,00005	**0,097**	**9600**	**0**	**13,28**	6609	**0,688**	**0,903**
256	0,0001	**0,185**	**115200**	**1000**	**1,12**	78371	**0,680**	**0,732**
512	0	**0,000**	**9600**	**0**	**10,52**	8344	**0,869**	**1,000**
512	0,00001	**0,040**	**9600**	**0**	**11,93**	7356	**0,766**	**0,960**
512	0,00005	**0,185**	**9600**	**0**	**14,71**	5966	**0,621**	**0,815**
512	0,0001	**0,336**	**9600**	**0**	**17,83**	4921	**0,513**	**0,664**



In general, our efficiency is close (80%) to the theoretical. This difference can be explained by the overhead of the logic implemented in the data link and application layers.

During our efficiency measurements we were also able to successfully test or protocol for different baud rates, frame lengths, BER and propagation times. These tests were already performed in the laboratory environment, as well as tests with different sized files and the brief disconnection of the cable, and they were also successful.

Conclusions

Through this project, we gained a deeper understanding of how to implement a data link layer and application layer, the Stop-and-Wait protocol, byte stuffing and error handling mechanisms. The implementation enhanced our comprehension of how data can be transmitted over serial connections while ensuring accuracy and reliability.

All major objectives were successfully achieved, resulting in an enjoyable and rewarding experience that fulfilled the project's intended goals.

Appendix I – Source Code

link_layer.c

```
// Link layer protocol implementation

#include "link_layer.h"
#include "serial_port.h"
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

// MISC
#define _POSIX_SOURCE 1 // POSIX compliant source


#define FLAG 0x7E      /* Synchronisation: start or end of frame */
#define A_TX 0x03      /* Address field in frames that are commands sent by the Transmitter or replies sent
by the Receiver */
#define A_RX 0x01      /* Address field in frames that are commands sent by the Receiver or replies sent by
the Transmitter */

// CONTROL FIELD to indicate the type of supervision frame/message
#define SET 0x03    /* SET frame: sent by the transmitter to initiate a connection */
#define UA 0x07        /* UA frame: confirmation to the reception of a valid supervision frame */
#define DISC 0x0B       /* DISC frame: to indicate the termination of a connection */
#define INF_0 0x00 // Information frame number 0
#define INF_1 0x80 // Information frame number 1, could be 0x40
#define INF_(n) ( (n) == 0 ? INF_0 : INF_1 )
#define ESC 0x7D // Byte stuffing escape octet
#define RR0 0xAA
#define RR1 0xAB
#define RR(n) ( (n) == 0 ? RR0 : RR1 )
#define REJ0 0x54
#define REJ1 0x55
#define REJ(n) ( (n) == 0 ? REJ0 : REJ1 )

#define NEXT_FRAME(f) ( ((f) + 1) % 2)


unsigned char frame_expected = 0;

// COMMS STATISTICS
typedef struct {
   unsigned int frames_sent;
   unsigned int frames_received;
   unsigned int frames_received_error;
   unsigned int frames_duplicated;
   unsigned int data_frames_sent;
   unsigned int data_frames_received;
   unsigned int retransmissions;
   unsigned int timeouts;
} comms_stats;

comms_stats stats = {0};

typedef enum {
            START_RCV,      /* Start of the receiving process */
            FLAG_OK,        /* Start flag ok */
            A_OK,           /* Address field ok */
   A_OK_RX,        /* Address field ok (commands from rx and replies from tx) */
            C_OK,           /* Control field ok */
            BCC_OK,         /* BCC field ok */
            STOP_RCV        /* Stop the receiving process */
} state_t;
```

```
LinkLayer connectionParams;


int frame_to_send = 0;


int alarmEnabled = FALSE;
int alarmCount = 0;

// Alarm function handler
void alarmHandler(int signal)
{
    alarmEnabled = FALSE;
    alarmCount++;

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

void enableAlarm(int time) {
    alarm(time);
    alarmEnabled = TRUE;
}

void clearAlarm() {
    alarm(0);
    alarmCount = 0;
}


void printStatistics() {
    printf("\n-------CONNECTION STATISTICS-------\n");
    printf("Number of total frames sent = %d\n", stats.frames_sent);
    printf("Number of total frames received = %d\n", stats.frames_received);
    printf("\n");
    if (connectionParams.role == LlTx)
        printf("Number of total data frames sent = %d\n", stats.data_frames_sent);
    if (connectionParams.role == LlRx) {
        printf("Number of total data frames received = %d\n", stats.data_frames_received);
        printf("Number of total duplicated data frames = %d\n", stats.frames_duplicated);
        printf("Number of total frames received with data errors = %d\n", stats.frames_received_error);
    }
    printf("\n");
    printf("Number of retransmissions = %d\n", stats.retransmissions);
    printf("Number of timeouts = %d\n", stats.timeouts);
}


int receiveSupervision(unsigned char a, unsigned char c, int timeout) {
    char byte;

    state_t state = START_RCV;

    while (state != STOP_RCV) {

        if (readByte(&byte) == 1) {

            switch (state) {

                case START_RCV:
                    if (byte == FLAG)
                        state = FLAG_OK;
                    break;

                case FLAG_OK:
                    if (byte == a)
                        state = A_OK;
```

```
                else if (byte != FLAG)
                    state = START_RCV;
                break;

            case A_OK:
                if (byte == FLAG)
                    state = FLAG_OK;
                else if (byte == c)
                    state = C_OK;
                else
                    state = START_RCV;
                break;

            case C_OK:
                if (byte == FLAG)
                    state = FLAG_OK;
                else if (byte == (a ^ c))
                    state = BCC_OK;
                else
                    state = START_RCV;
                break;

            case BCC_OK:
                if (byte == FLAG)
                    state = STOP_RCV;
                else
                    state = START_RCV;
                break;

            default:
                printf("ERROR: Wrong state (%d) in 'receiving supervision frame' state machine", state);
                break;
        }

    }
    // Timeout only if sender is waiting for acknowledgement
    if (timeout == TRUE && alarmEnabled == FALSE) {
        stats.timeouts++;
        return -1;
    }
        }

    return 0;
}

int sendSupervision(unsigned char a, unsigned char c) {
    // Create frame to send
    char frame[5] = {0};

    frame[0] = FLAG;
    frame[1] = a;
    frame[2] = c;
    frame[3] = a ^ c;
    frame[4] = FLAG;

    // Write the frame until all 5 bytes are written
    while (writeBytes(frame, 5) != 5);

    return 0;
}

////////////////////////////////////////////////
// LLOPEN
////////////////////////////////////////////////
int llopen(LinkLayer connectionParameters)
{
    int dl_identifier = openSerialPort(connectionParameters.serialPort, connectionParameters.baudRate);
```

```
    if (dl_identifier < 0) return -1;

    connectionParams = connectionParameters;

    (void)signal(SIGALRM, alarmHandler);

    int retransmissions = connectionParameters.nRetransmissions;

    switch (connectionParameters.role)
    {

    case LlTx:
        while (alarmCount <= retransmissions){
            sendSupervision(A_TX,SET);
            enableAlarm(connectionParameters.timeout);
            stats.frames_sent++;

            // It's a retransmission
            if (alarmCount != 0) {
                stats.retransmissions++;
            }


            if (receiveSupervision(A_RX, UA, TRUE) == 0) {
                stats.frames_received++;
                clearAlarm();
                printf("Successfully connected!\n");
                return 0;
            }

        }

        // Cancel the procedure, maximum number of retransmissions exceeded
        clearAlarm();
        printf("Maximum number of retransmissions exceeded!\n");
        return -1;

    case LlRx:
        while (1){

            if (receiveSupervision(A_TX, SET,0) == 0) {
                stats.frames_received++;
                sendSupervision(A_RX,UA);
                stats.frames_sent++;
                printf("Successfully connected!\n");
                break;
            }

            return -1;
        }
        break;

    default:
        return -1;
        break;
    }

    return dl_identifier;
}


int prepare_frame(char *f_buf, const unsigned char *buf, int bufSize) {
    f_buf[0] = FLAG;
    f_buf[1] = A_TX;
    f_buf[2] = INF_(frame_to_send);
    f_buf[3] = f_buf[1] ^ f_buf[2];
```

```
    int num_bytes = 4;
    char bcc2 = 0;

    // Data packet
    for (int i = 0; i < bufSize; i++, num_bytes++) {

        if (buf[i] == FLAG || buf[i] == ESC) {
            // Data byte needs to be stuffed
            f_buf[num_bytes++] = ESC;
            f_buf[num_bytes] = buf[i] ^ 0x20;
        } else {
            f_buf[num_bytes] = buf[i];
        }

        bcc2 ^= buf[i];
    }

    if (bcc2 == FLAG || bcc2 == ESC) {
        // BCC2 Needs to be stuffed
        f_buf[num_bytes++] = ESC;
        f_buf[num_bytes++] = bcc2 ^ 0x20;
    } else {
        f_buf[num_bytes++] = bcc2;
    }

    f_buf[num_bytes++] = FLAG;

    return num_bytes;
}

// Returns 0 if acknowledged, 1 if rejected and 2 if timeout
int waitWriteResponse() {
    state_t state = START_RCV;
    unsigned char c = 0;

    while (state != STOP_RCV) {
        unsigned char byte;

        if ( readByte(&byte) == 1 ) {

            switch (state) {

                case START_RCV:
                    if (byte == FLAG)
                        state = FLAG_OK;
                    break;

                case FLAG_OK:
                    if (byte == A_TX)
                        state = A_OK;
                    else if (byte != FLAG)
                        state = START_RCV;
                    break;

                case A_OK:
                    if (byte == FLAG)
                        state = FLAG_OK;
                    else if ( (byte == REJ(frame_to_send)) | (byte == RR(NEXT_FRAME(frame_to_send))) ) {
                        c = byte;
                        state = C_OK;
                    } else
                        state = START_RCV;
                    break;

                case C_OK:
                    if (byte == FLAG)
                        state = FLAG_OK;
```

```
                else if (byte == (A_TX ^ c))
                    state = BCC_OK;
                else
                    state = START_RCV;
                break;

            case BCC_OK:
                if (byte == FLAG)
                    state = STOP_RCV;
                else
                    state = START_RCV;
                break;

            default:
                printf("ERROR: Wrong state (%d) in 'receiving supervision frame' state machine", state);
                break;
        }
    }

    if (alarmEnabled == FALSE) {
        // Timeout
        stats.timeouts++;
        return 2;
    }
  }

  if (c == RR(NEXT_FRAME(frame_to_send)))
    return 0;
  return 1;
}


////////////////////////////////////////////////
// LLWRITE
////////////////////////////////////////////////
int llwrite(const unsigned char *buf, int bufSize)
{
    // Frame buffer, data bytes (bufSize) + 4 other field bytes + 2 flags
    // If all bytes are stuffed then number of bytes is doubled with the exception of the start and end flags
    unsigned char f_buf[2 *(bufSize + 4) + 2];

    int frameSize = prepare_frame(f_buf, buf, bufSize);

    int transmissions = 0;

    while (alarmCount <= connectionParams.nRetransmissions) {
        // Send frame
        if (writeBytes(f_buf, frameSize) != frameSize) {
            printf("ERROR: writeBytes() didn't write all bytes\n");
            continue; // so para quando escrever
        }
        enableAlarm(connectionParams.timeout);
        stats.frames_sent++;
        stats.data_frames_sent++;
        transmissions++;

        // It's a retransmission
        if (transmissions > 1)
            stats.retransmissions++;

        // Wait for response
        int response = waitWriteResponse();
        stats.frames_received++;
        if (response == 0) {
            // Frame successfully akcnowledged
            clearAlarm();
            frame_to_send = NEXT_FRAME(frame_to_send);
```

```
            return bufSize;
        } else  if (response == 1) {
            // Frame rejected
            clearAlarm();
        }
    }

    printf("Maximum number of retransmissions exceeded!\n");
    clearAlarm();
    return -1;
}

////////////////////////////////////////////////
// LLREAD
////////////////////////////////////////////////
int llread(unsigned char *packet)
{
    unsigned char byte,c_byte;
    state_t state = START_RCV;
    int char_read = 0;
    unsigned char escape_next=0;
    unsigned char bcc = 0;

    while (state!= STOP_RCV){

        if (readByte(&byte) == 1) {

            switch (state)
            {
            case START_RCV:
                if (byte == FLAG)
                    state = FLAG_OK;
                break;

            case FLAG_OK:

                if (byte == A_TX){
                    state = A_OK;
                }
                else if (byte !=FLAG)
                    state = START_RCV;
                break;

            case A_OK:

                if (byte == INF_0 || byte == INF_1){
                    c_byte = byte;
                    byte = byte >> 7;
                    if (byte == frame_expected){
                        state = C_OK;
                    }
                    else{
                        sendSupervision(A_TX, RR(frame_expected));
                        stats.frames_received++; // duplicate or out of order
                        stats.frames_sent++;
                        stats.frames_duplicated++;
                        stats.data_frames_received++;
                        state = START_RCV;
                    }
                }
                else if (byte == FLAG)
                    state = FLAG_OK;
                else
                    state = START_RCV;
                break;

            case C_OK:
```

```
            if (byte == (A_TX ^ c_byte)){
                state = BCC_OK;
            }
            else if (byte == FLAG)
                state = FLAG_OK;
            else
                state = START_RCV;
            break;

        case BCC_OK:
            if (byte == FLAG) {
                int i = char_read - 1;
                unsigned char bcc2 = packet[i--];

                while (i>=0){
                    bcc ^= packet[i--];

                }
                if (bcc == bcc2){

                    frame_expected ^= 0x01; //want to receive next packet
                    sendSupervision(A_TX, RR(frame_expected));
                    stats.frames_received++;
                    stats.frames_sent++;
                    stats.data_frames_received++;
                    state = STOP_RCV;
                    break;
                }
                else{
                    if(frame_expected) {
                        sendSupervision(A_TX, REJ1);
                    } else {
                        sendSupervision (A_TX,REJ0);
                    }
                    char_read = 0;
                    bcc = 0;
                    stats.frames_received++;
                    stats.frames_received_error++;
                    stats.data_frames_received++;
                    stats.frames_sent++;
                    state = START_RCV;
                    break;
                }

            } else if (byte == ESC) {
                escape_next = 1;
            } else {
                if (escape_next) {
                    byte ^= 0x20;
                    escape_next = 0;
                }
                packet[char_read++] = byte;
            }
            break;

        default:
            printf("Wrong state");
            break;
        }
      }
    }

    return char_read;
}


int waitDiscResponse() {
```

```
state_t state = START_RCV;
char c = 0;
char a = 0;

while (state != STOP_RCV) {
   char byte;

   if ( readByte(&byte) == 1 ) {
      switch (state) {

         case START_RCV:
            if (byte == FLAG)
               state = FLAG_OK;
            break;

         case FLAG_OK:
            if (byte == A_TX) {
               a = byte;
               state = A_OK;
            } else if (byte == A_RX) {
               a = byte;
               state = A_OK_RX;
            } else if (byte != FLAG)
               state = START_RCV;
            break;

         case A_OK:
            if (byte == FLAG)
               state = FLAG_OK;
            else if (byte == DISC) {
               c = byte;
               state = C_OK;
            } else
               state = START_RCV;
            break;

         case A_OK_RX:
            if (byte == FLAG)
               state = FLAG_OK;
            else if (byte == UA) {
               c = byte;
               state = C_OK;
            } else
               state = START_RCV;
            break;

         case C_OK:
            if (byte == FLAG)
               state = FLAG_OK;
            else if (byte == (a ^ c))
               state = BCC_OK;
            else
               state = START_RCV;
            break;

         case BCC_OK:
            if (byte == FLAG)
               state = STOP_RCV;
            else
               state = START_RCV;
            break;

         default:
            printf("ERROR: Wrong state (%d) in 'receiving supervision frame' state machine", state);
            break;
      }
   }
```

```
        if (alarmEnabled == FALSE) {
            // Timeout
            stats.timeouts++;
            return 2;
        }
    }

    if (c == UA)
        return 0;
    return 1;
}



////////////////////////////////////////////////
// LLCLOSE
////////////////////////////////////////////////
int llclose(int showStatistics)
{
    if (connectionParams.role == LITx) {

        while (TRUE) {

            // Send DISC frame
            sendSupervision(A_TX, DISC);
            enableAlarm(connectionParams.timeout);
            stats.frames_sent++;

            // Successfully receives DISC
            if (receiveSupervision(A_RX, DISC, TRUE) == 0) {
                stats.frames_received++;
                clearAlarm();
                sendSupervision(A_RX, UA);
                stats.frames_sent++;
                usleep(1000);
                printf("Successfully disconnected!\n");
                break;
            }

            // Cancel the procedure, maximum number of retransmissions exceeded
            if (alarmCount > connectionParams.nRetransmissions) {
                clearAlarm();
                printf("Maximum number of retransmissions exceeded!\n");
                return -1;
            }

            stats.retransmissions++;
        }

    } else if (connectionParams.role == LIRx) {

        receiveSupervision(A_TX, DISC, FALSE);
        stats.frames_received++;

        while (TRUE) {

            // Send DISC frame
            sendSupervision(A_RX, DISC);
            enableAlarm(connectionParams.timeout);
            stats.frames_sent++;

            int response = waitDiscResponse();
            if (response == 0) {
                // Successfully receives UA
                stats.frames_received++;
```

```
            clearAlarm();
            printf("Successfully disconnected!\n");
            break;
        } else  if (response == 1) {
            // DISC received
            clearAlarm();
        }

        // Cancel the procedure, maximum number of retransmissions exceeded
        if (alarmCount > connectionParams.nRetransmissions) {
            clearAlarm();
            printf("Maximum number of retransmissions exceeded!\n");

            closeSerialPort(); // deve fechar

            return -1;
        }

        stats.retransmissions++;
    }

}

int clstat = closeSerialPort();

if (showStatistics == TRUE)
    printStatistics();

return clstat;
}
```

application_layer.c

```
// Application layer protocol implementation

#include "application_layer.h"
#include "link_layer.h"
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>

#include <stdio.h>

#define PCKT_C_START 0x1
#define PCKT_C_DATA 0x2
#define PCKT_C_END 0x3
#define PCKT_T_FILE_SZ 0x0
#define PCKT_T_FILE_NM 0x1

#define FRAGMENT_SZ 512
#define DATA_PCKT_SZ (FRAGMENT_SZ + 4)
#define PCKT_L2 (FRAGMENT_SZ / 256)
#define PCKT_L1 (FRAGMENT_SZ % 256)



void applicationLayer(const char *serialPort, const char *role, int baudRate,
                      int nTries, int timeout, const char *filename)
{
    LinkLayer connectionParameters = {
        .baudRate = baudRate,
        .nRetransmissions = nTries,
```

```
    .timeout = timeout
};
strncpy(connectionParameters.serialPort,
        serialPort,
        sizeof(connectionParameters.serialPort) - 1);

// RECEIVER --------------------
if (!strcmp(role, "rx")){
    connectionParameters.role = LlRx;

    llopen(connectionParameters);

    unsigned char start_pckt[256] = {0};

    unsigned int file_sz = 0;
    unsigned char filename_tx[256] = {0};

    // START PACKET RECEIVE
    while (TRUE){
        if (llread(start_pckt) < 1){
            continue;
        }
        if (start_pckt[0] == PCKT_C_START){
            if (start_pckt[1] == PCKT_T_FILE_SZ){
                unsigned int filename_size_sz = start_pckt[2];
                memcpy(&file_sz, &start_pckt[3], filename_size_sz);
                if (start_pckt[3 + filename_size_sz] == PCKT_T_FILE_NM){
                    memcpy(filename_tx,
                    &start_pckt[5 + filename_size_sz],
                    start_pckt[4 + filename_size_sz]);
                }
            }
            break;
        }
    }


    // Filename given by transmitter or given by argument in receiver?
    //int fd_target = open(filename_tx, O_WRONLY | O_CREAT, S_IRUSR);
    int fd_target = open(filename, O_WRONLY | O_CREAT, 0666);

    int bytes_read = 0;
    unsigned char data_pckt[DATA_PCKT_SZ] = {0};
    int seq_n = -1;
    unsigned char l2, l1 = 0;

    // DATA PACKETS RECEIVE
    while (TRUE){
        llread(data_pckt);

        if ((data_pckt[0] == PCKT_C_DATA)
                &&
                (data_pckt[1] == (seq_n + 1)%100)){
            seq_n = data_pckt[1];
            l2 = data_pckt[2];
            l1 = data_pckt[3];
            bytes_read += write(fd_target,
                            &data_pckt[4],
                            l2*256+l1);
        }
        // END PACKET RECEIVE
        else if (data_pckt[0] == PCKT_C_END)
        {
            if (bytes_read != file_sz)
                printf("ERROR: File is not complete\n");
            break;
        }
```

```
    }

    close(fd_target);
    llclose(TRUE);

}
// TRANSMITTER ------------------
else if (!strcmp(role, "tx")){
    connectionParameters.role = LlTx;
    int fd_data = open(filename, O_RDONLY);

    llopen(connectionParameters);

    struct stat st;
    stat(filename, &st);
    int file_sz = st.st_size;
    int filename_sz = strlen(filename);

    // START PACKET ASSEMBLY
    unsigned char ctrl_pckt[256] = {0};
    ctrl_pckt[0] = PCKT_C_START;
    ctrl_pckt[1] = PCKT_T_FILE_SZ;
    ctrl_pckt[2] = sizeof(int);
    memcpy(&ctrl_pckt[3], &file_sz, sizeof(int));
    ctrl_pckt[7] = PCKT_T_FILE_NM;
    ctrl_pckt[8] = filename_sz;
    memcpy(&ctrl_pckt[9], filename, filename_sz);

    if (!llwrite(ctrl_pckt, 5 + sizeof(int) + filename_sz)){
        return;
    }

    // DATA PACKETS ASSEMBLY
    int n_fragments = (file_sz + FRAGMENT_SZ - 1) / FRAGMENT_SZ;
    unsigned char data_pckt[DATA_PCKT_SZ] = {0};
    int n = 0;
    while (n < n_fragments){
        memset(data_pckt, 0, DATA_PCKT_SZ);
        data_pckt[0] = PCKT_C_DATA;
        int fragment_sz = read(fd_data, &data_pckt[4], FRAGMENT_SZ);

        data_pckt[1] = n % 100;
        data_pckt[2] = fragment_sz / 256;
        data_pckt[3] = fragment_sz % 256;

        n++;
        llwrite(data_pckt,4+fragment_sz);

    }

    // END PACKET ASSEMBLY
    ctrl_pckt[0] = PCKT_C_END;
    ctrl_pckt[1] = PCKT_T_FILE_SZ;
    ctrl_pckt[2] = sizeof(int);
    memcpy(&ctrl_pckt[3], &file_sz, sizeof(int));
    ctrl_pckt[7] = PCKT_T_FILE_NM;
    ctrl_pckt[8] = filename_sz;
    memcpy(&ctrl_pckt[9], &filename, filename_sz);

    if (!llwrite(ctrl_pckt, 5 + sizeof(int) + filename_sz)){
        return;
    }

    close(fd_data);
    llclose(TRUE);
```

```
    }
}
```