U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Data Link Protocol

1st Lab Work Report RCom - 1st Semester 24/25

Afonso Castro - up202208026
Leonor Couto - up202205796

Summary

This document is a report of our semester's first laboratorial project which addresses the development of a data link layer protocol for file transfer between two computers. We designed the protocol to effectively manage both the data link and application layers, ensuring good communication between the two roles: transmitter and receiver. The architecture was structured to allow each side to perform its tasks independently, yet at the same time. In this report you will find information about the design, implementation, testing, and validation processes involved in building the protocol.

The Link layer managed low-level serial communication, while the Application layer handled file data and coordinated the transfer process. This project improved our understanding of how communication protocols work, and we learned how they can be adapted to specific environments and needs. The testing and validation phases allowed us to improve the code and its reliability, in that way improving our problem-solving and project development skills.

Introduction

For this semester's first laboratorial work, it was proposed that we create a system where we implement a data link protocol that would allow us to transfer a certain file between computers connected by a RS-232 serial cable.

In order for us to continue building and testing our project in times where we didnt have the access to two computers, we used a program that allowed us to use two virtual serial ports, in that way simulating on one machine the serial cable connection that would occur between two machines.

The goal of this report is to document, explain and inform about the various sections of our data link layer protocol implementation, in order to help users understand our work and the environment of the project. Knowing this, this report is organized in the following sections:

- **Architecture** - This section details the functional blocks and interfaces that constitute the protocol's framework, providing a foundation for understanding the system's structure.
- **Code Structure** - Here, we dive into the APIs, data structures, and main functions, showing how each component aligns with the architecture to allow the protocol to function correctly.
- **Main Use Cases** - This section identifies the principal use cases and describes the sequences of function calls involved in our code.
- **Logical Link Protocol** - We highlight the main functional aspects of the logical link layer, explaining the strategy used in its implementation with relevant code examples.
- **Application Protocol** - This section focuses on the application layer's main functions and also showcases its implementation with code snippets.
- **Validation** - Here we present the tests we conducted on the protocol.
- **Data Link Protocol Efficiency** - This section provides a statistical assessment of the protocol's efficiency, supported by measurements from the developed code.

Architecture

The architecture of our project is created to handle both the roles of the **transmitter** and the **receiver** in our data transfer scenario, meaning that both the transmitter and receiver code are contained within the same files. The project is structured to ensure that each role only executes the code relevant to its function, preventing any code designated for one role from running on the other.

In addition to this role division, the project is organized into two main layers: the **Application Layer** and the **Link Layer**. Each layer has distinct responsibilities, with the Application Layer managing high-level data handling and the Link Layer handling the lower-level data transmission over the serial port connection.

In order to help the Link Layer work correctly, the already pre-provided **serial port interface functions** allow our Link Layer code to successfully open, close, read and write in the serial port.

Code Structure

To achieve our project's goals, we developed and used various functions and structs, ensuring they followed a clear and understandable design.

In the Application Layer (*application_layer.c*) we did not create any data structures and the only API used was the Link Layer (*link_layer.c*). The functions that were built and implemented in this file were the following :

```
	//This function creates the control packet
	unsigned char * createControlPacket(const char* filename, int fileSize, unsigned int* controlPacketSize, const unsigned int cField)
	//This function creates the data packet
	unsigned char * createDataPacket(unsigned char* payload, int payloadSize, unsigned int* dataPacketSize, unsigned char sequenceNumber)
	//This function creates the control packet
	void analyseControlPacket(unsigned char* packet, int sizePacket, unsigned char* receivedFilename)
	//Main function that controls all the others
	void applicationLayer(const char *serialPort, const char *role, int baudRate, int nTries, int timeout, const char *filename)
```

In the Link Layer (*link_layer.c*) we used three data structures, two of them were provided, but the last was created by us. ***LinkLayerRole*** identifies if the computer is the transmitter or the receiver, ***LinkLayer*** is where the parameters associated with the transfer of data and the opening of the serial port are saved, and ***state_t*** identifies the state of the messages while being received. This file has an API that is the Serial Port (*serial_port.c*).

```
typedef enum {
	LlTx,
	LlRx,
} LinkLayerRole;

typedef struct {
char serialPort[50];
LinkLayerRole role;
int baudRate;
int nRetransmissions;
int timeout;
} LinkLayer;
```

```
typedef enum {
START_STATE,
FLAG_STATE,
A_STATE,
C_STATE,
BCC_STATE,
STOP_STATE,
DESTUFFING_STATEZ
} state_t;
```

Also in this layer we have four main functions that together manage all the data transfer and serial port communication operations: **llopen**, **llwrite**, **llread** and **llclose**.

Additionally, this layer includes two extra functions: **alarmHandler** for activating alarms and another, **resendUA**, which ensures the receiver resends the UA (acknowledgement) if the transmitter hasn't received it.

```
        //This function creates the alarm
        void alarmHandler(int signal)
        //This function ensures the receiver resends the UA if the transmitter
hadn't received it
        void resendUA()
        //This function establishes the connection between the computers
        int llopen(LinkLayer connectionParameters)
        //This function sends the frames
        int llwrite(const unsigned char *buf, int bufSize)
        //This function receives the frames
        int llread(unsigned char *packet)
        //This function closes the connection
        int llclose(int showStatistics)
```

Main Use Cases

As mentioned earlier, since the computer running the code is either the transmitter or the receiver, the specific functions it calls and runs will differ based on its role.

The most important functions that each role calls are presented next:

Transmitter

1. **llopen():** Ensures communication between the computers through the serial port is established.
2. **createControlPacket():** Creates the control packets.
3. **createDataPacket():** Creates the data packets.
4. **llwrite():** Creates and sends the information frame containing the packets through the serial port.
5. **llclose():** Closes the serial port connection.

Receiver

1. **llopen():** Ensures communication between the computers through the serial port is established.
2. **analyseControlPacket():** Analyzes the content of the control packets.
3. **llread():** Receives the information frame containing the packets through the serial port.
4. **llclose():** Closes the serial port connection.

Logical Link Protocol

The Logical Link layer is the one responsible for interacting with the Serial Port, being responsible for the communication between the transmitter and the receiver. We used the STOP-and-WAIT protocol to allow the communication between the computers. This layer has four main functions, as mentioned before: llopen, llwrite, llread and llclose.

Starting with llopen, this function initiates communication between the two computers, ensuring the successful transmission of predefined messages. The transmitter sends a message indicating it wants to start a "conversation", and the receiver responds with an acknowledgment. This exchange confirms that both the transmitter and receiver are ready to read and write messages.

With the two machines now communicating, the llwrite and llread functions work in parallel: llwrite is used by the transmitter and llread is used by the receiver.

The llwrite function begins by preparing the messages to be sent. Here, it constructs the frame, including the flag, address, control, BCC1 and BCC2 fields, as well as the information from the packets received from the Application Layer. The frame is then bit-stuffed to prevent the receiver from misinterpreting any flag within the message data as the end of the message.

```
// llwrite - stuffing and sending frame
if(BCC2 == FLAG){ // If 0x7e -> 0x7d 0x5e
 frame = realloc(frame,++sizeOfFrame);
 frame[k] = ESCAPE;
 k++;
 frame[k] = XOR_FLAG;
 k++;
} else if (BCC2 == ESCAPE) {  // If 0x7d -> 0x7d 0x5dc
 frame = realloc(frame,++sizeOfFrame);
 frame[k] = ESCAPE;
 k++;
 frame[k] = XOR_ESCAPE;
 k++;
} else {
 frame[k] = BCC2;
 k++;
}
frame[k] = FLAG; // Flag
k++;

printf("------------------------------------------------------------------------\n");
    printf("Write I (llwrite)\n");
    int bytesW = writeBytesSerialPort(frame, k);
    printf("%d bytes written (llwrite)\n", bytesW);

printf("------------------------------------------------------------------------\n");
```

With the prepared frame, the transmitter can now send it and wait for confirmation from the receiver. If the confirmation is a rejection (REJ), the transmitter resends the frame and waits again. If the response is a positive acknowledgment (RR), the transmitter proceeds to send the next prepared frame.

Meanwhile, the llread function operates on the Receiver's side to check the incoming frames. It starts by receiving the frame, destuffing it, and verifying its correctness, including the BCC2 check. If the frame is correct, the Receiver sends a positive acknowledgment (RR), indicating it is ready to receive more frames. If there is an issue with the frame, the Receiver sends a rejection (REJ) message.

```
// llread - destuffing
if(byte == XOR_ESCAPE){
 packet[i++] = ESCAPE;
 stateR = BCC_STATE;
} else if (byte == XOR_FLAG) {
 packet[i++] = FLAG;
 stateR = BCC_STATE;
} else {
 stateR = STOP_STATE;
 return -1;
}
```

The last main function in the Link Layer is llclose, which has the simple goal of ending the communication between the transmitter and receiver. It starts with the transmitter sending a "disconnect" message (DISC). The receiver receives this message, responds with a similar DISC message, and the process concludes with a final acknowledgment sent by the transmitter to the receiver. Both sides then close the connection.

At the end of llclose, both the transmitter and receiver print statistics related to the performance of the code each role executed.

Both llclose and llopen maintain a clear separation of roles within each function, ensuring that the transmitter never executes code intended for the receiver, and vice versa.

Application Protocol

The Application Layer interacts with the user and handles the actual file being transferred. Like certain functions in the Link Layer, it has a division between the transmitter and receiver roles, ensuring that each role only executes the code specific to its function. The Link Layer API manages the actual file transfer, translating the data packets provided by the Application Layer into information frames.

From the Transmitter's perspective, the Application Layer breaks down the file into smaller data chunks, which form the content of the frames that the Link Layer will send. On the Receiver's side, the Application Layer includes the code necessary to analyze these incoming frames and reconstruct the file.

This function starts with the call of llopen, even before separating by roles, because both the transmitter and the receiver have to execute this function to ensure the start of the communication.

After this, the code branches according to the roles.

The transmitter starts by opening the file, then creates and sends the starting control packet to signal the beginning of the file transfer via call to llwrite. Once the control packet is sent, the transmitter proceeds to send the data packets, which contain the chunks of the file.

```
// applicationLayer - loop for sending the data packets
while (bytesRemaining > 0) {

       int payloadSize=0;
       if (bytesRemaining > (long int) MAX_PAYLOAD_SIZE){
        payloadSize = MAX_PAYLOAD_SIZE;
       } else {
        payloadSize = bytesRemaining;
       }

       unsigned char* payload = (unsigned char* ) malloc(payloadSize); // Allocate
memory for payload.
       memcpy(payload, content, payloadSize); // Copy payload from content buffer.

       unsigned int dataPacketSize; // Data packet size.
       unsigned   char   *dataPacket   =   createDataPacket(payload,   payloadSize,
&dataPacketSize, sequenceNum);
```

```
        if (llwrite(dataPacket, dataPacketSize) == -1) {
         printf("ERROR: llwrite data packet failed");
         return;
        }

        sequenceNum = (sequenceNum+1) % 255;

        bytesRemaining = bytesRemaining - payloadSize;
                       content += payloadSize;
}
```

After all the file's data has been sent, the transmitter sends the ending control packet, signaling that all data packets have been delivered and marking the end of the file transfer.

On the receiver side the process is similar but for receiving data via call to llread.

First it receives the starting control packet and only then opens a file where he stores the data that starts reading from the incoming packets. The final packet received is the ending control packet, indicating that the entire file has been transferred.

Both the transmitter and the receiver then call Link Layer's llclose to end the connection.

Validation

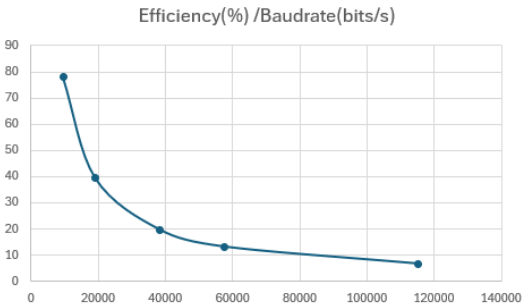
In order to check the correctness and efficiency of our project, we elaborated tests in order to observe the code's behavior:

- Transfer of different files with different sizes and different file names.
- Transfer of files with different baudrates.
- Transfer of different data packet sizes.
- Transfer with total and partial interruptions of the serial port.
- Transfer with noise between the serial ports.

Data Link Protocol Efficiency

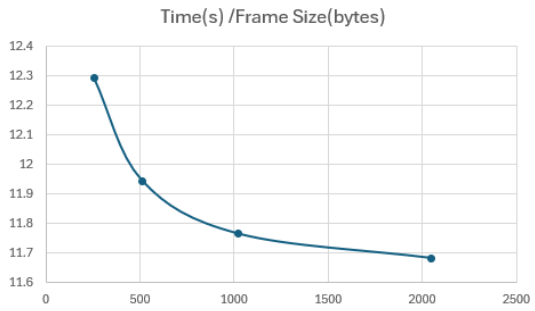
Baudrate variation

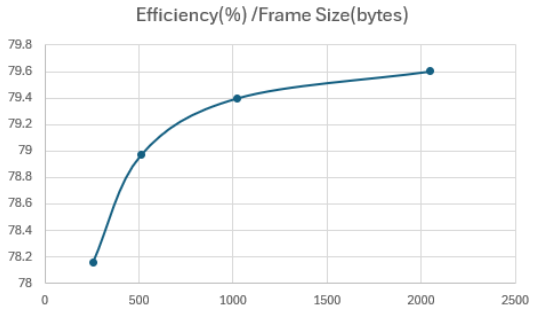
With the penguin.gif file and the size of the information frames fixed at 1000, we were able to understand that the efficiency of the protocol reduces with the increase of the baudrate, because even though the velocity of the propagation of the frames increases, the time of transmission of said information also increases.



Frame size variation

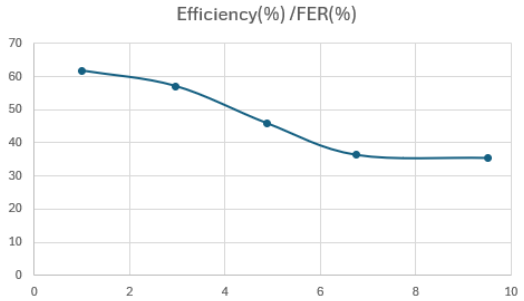
With the penguin.gif file and baudrate fixed at 9600 bits/s, we were able to conclude that the total time of the transfer and the size of the frame are inversionally proportional, because if the frame its smaller we have to send more frames and that results in more time. We were also able to conclude that when we increase the frame size the efficiency also increases.





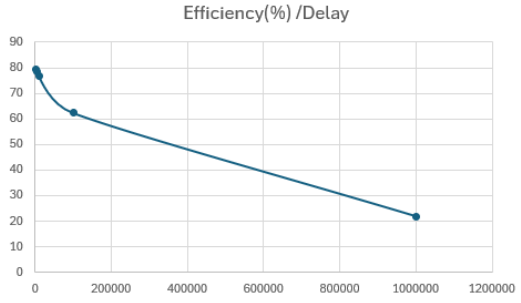
Frame Error Ratio variation

This was tested with the penguin.gif file, baudrate fixed at 9600 bits/s and the size of the information frames fixed at 1000. In the Stop-and-Wait protocol, one error in the frame promotes the retransmission and increases the total time used. So, the efficiency is inversionally proportional to the error rate, caused by all the retransmissions needed.



Propagation delay variation

This was tested with the penguin.gif file and baudrate fixed at 9600 bits/s and the size of the information frames fixed at 1000. When we increase the propagation delay we can conclude that the efficiency reduces, because when we create this delay it will naturally take more time, reducing the efficiency.



Conclusions

In this project, we successfully designed and implemented a data link layer protocol for file transfer between two computers connected by a RS-232 serial cable. The project involved creating both the protocol's Logical Link and Application layers, and managing the communication between the transmitter and receiver, ensuring an efficient data transmission.

The architecture of the system was designed to separate the roles of the transmitter and receiver, with distinct responsibilities assigned to each role, ensuring that both parts of the system operated independently but at the same time.

The Link layer handled the low-level serial communication and error detection, while the Application layer was responsible for managing the file data and coordinating the start and end of the data transfer.

With this being said, this project helped us gain a better understanding of the characteristics and functionality of data link protocols, including their architecture and logic. It also demonstrated how communication protocols can be adapted to different environments and situations.

Additionally, the multiple efforts and brainstorming that occurred from testing the program during and after the development of the code also improved our problem-solving skills and code quality.

Appendix I - Source Code

link_layer.c

```
// Link layer protocol implementation

#include "../include/link_layer.h"
#include "../include/serial_port.h"
#include <signal.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <termios.h>
#include <unistd.h>
#include <time.h>

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

#define BUF_SIZE 5

#define FLAG 0x7E
#define ADDRESS_SEND 0x03
#define CONTROL_SET 0x03

#define ADDRESS_RECEIVE 0x01
```

```c
#define CONTROL_UA 0x07

#define ESCAPE 0x7D
#define XOR_FLAG 0x5E
#define XOR_ESCAPE 0x5D


int alarmEnabled = FALSE;
int alarmCount = 0;
int retransmitionsCount = 0;

int byteCount = 0;
int alarmTotalAlarmCount = 0;

unsigned char lastFrame[5];


double total_time_used;
double time_used;
clock_t startTotal, endTotal;
clock_t start, end;

typedef enum {
    START_STATE,
    FLAG_STATE,
    A_STATE,
    C_STATE,
    BCC_STATE,
    STOP_STATE,
    DESTUFFING_STATE
} state_t;

LinkLayer connectionParametersCopy;

int Ns_t = 0;  // Transmiter
int Nr_r = 1; // Receiver
#define CONTROL_I_N0 0x00
#define CONTROL_I_N1 0x80
#define CONTROL_RR0  0xAA
#define CONTROL_RR1  0xAB
#define CONTROL_REJ0 0x54
#define CONTROL_REJ1 0x55

#define CONTROL_DISC 0x0B

int frameCounter = 0;
int frameCounterReceived = 0;

int counterNotSets = 0;



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

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


// Auxiliar function no resent the UA in case of necessity
void resendUA(){
    // READ WAITS FOR SET
    state_t stateR = C_STATE;
    int a_prov2 = ADDRESS_SEND;
    int c_prov2 = CONTROL_SET;

    while(stateR != STOP_STATE) {
        unsigned char byte;
        if(readByteSerialPort(&byte)){
            byteCount++;
            switch (stateR) {
                case C_STATE:
                    {if(byte == FLAG) {
                        stateR = FLAG_STATE;
                    } else if (byte == (a_prov2 ^ c_prov2)) {
                        stateR = BCC_STATE;
                    } else {
                        stateR = STOP_STATE;
                    }
                    break;}
                case BCC_STATE:
                    {if(byte == FLAG) {
                        stateR = STOP_STATE;
                        frameCounterReceived++;
                        //READ RESPONDE DE VOLTA
                        unsigned char bufR2[BUF_SIZE];
                               unsigned char BCC1R = ADDRESS_RECEIVE ^
CONTROL_UA;
                        bufR2[0] = FLAG;
                        bufR2[1] = ADDRESS_RECEIVE; //0X01
                        bufR2[2] = CONTROL_UA; //0X07
                        bufR2[3] = BCC1R;
                        bufR2[4] = FLAG;

printf("-----------------------------------------------------------------
---------\n");
                                printf("Read UA again (llopen -> resendUA)
\n");
                                 printf("Flag: 0x%02X | Address: 0x%02X |
Control: 0x%02X | BCC: 0x%02X | Flag: 0x%02X\n", bufR2[0], bufR2[1],
bufR2[2], bufR2[3], bufR2[4]);
                                 int bytesR = writeBytesSerialPort(bufR2,
BUF_SIZE);
                                printf("%d bytes written (UA)(llopen)\n",
bytesR);

printf("-----------------------------------------------------------------
---------\n");
                        frameCounter++;
                    } else {
                        stateR = STOP_STATE;
                    }
```

```
                    break;}
                default:
                    {break;}
                }
        }

    }

}



////////////////////////////////////////////////
// LLOPEN
////////////////////////////////////////////////
int llopen(LinkLayer connectionParameters) {

          int   fd   =   openSerialPort(connectionParameters.serialPort,
connectionParameters.baudRate);
    if (fd < 0) { return -1;}
    connectionParametersCopy.role = connectionParameters.role;
    connectionParametersCopy.baudRate = connectionParameters.baudRate;
                        connectionParametersCopy.nRetransmissions      =
connectionParameters.nRetransmissions;
    connectionParametersCopy.timeout = connectionParameters.timeout;

    switch (connectionParameters.role) {

        case LlTx:
        {
            //WRITE WRITES INICIAL SET
            unsigned char bufW[BUF_SIZE];

            unsigned char BCC1W = ADDRESS_SEND ^ CONTROL_SET;
            bufW[0] = FLAG;
            bufW[1] = ADDRESS_SEND; //0X03
            bufW[2] = CONTROL_SET; //0X03
            bufW[3] = BCC1W;
            bufW[4] = FLAG;

printf("-------------------------------------------------------------------
---------\n");
            printf("Write SET (llopen):\n");
             printf("Flag: 0x%02X | Address: 0x%02X | Control: 0x%02X |
BCC:  0x%02X  |  Flag:  0x%02X\n",  bufW[0],  bufW[1],  bufW[2],  bufW[3],
bufW[4]);
            int bytesW = writeBytesSerialPort(bufW, BUF_SIZE);
            printf("%d bytes written (SET)(llopen)\n", bytesW);

printf("-------------------------------------------------------------------
---------\n");
            frameCounter++;


            //WRITE WAITS FOR UA
            (void)signal(SIGALRM, alarmHandler);
            alarmEnabled = FALSE;
            alarmCount = 0;
```

```
            unsigned char bufW2[BUF_SIZE + 1] = {0}; // +1: Save space
for the final '\0' char
            state_t stateW = START_STATE;
            int a_prov1 = 0;
            int c_prov1 = 0;

                while(stateW != STOP_STATE && retransmitionsCount <
connectionParameters.nRetransmissions) {

                if (alarmEnabled == FALSE) {
                    if(alarmCount != 0){
                        //WRITE REWRITES SET
                        unsigned char bufW[BUF_SIZE];
                                unsigned char BCC1W = ADDRESS_SEND ^
CONTROL_SET;
                        bufW[0] = FLAG;
                        bufW[1] = ADDRESS_SEND; //0X03
                        bufW[2] = CONTROL_SET; //0X03
                        bufW[3] = BCC1W;
                        bufW[4] = FLAG;

printf("-----------------------------------------------------------------
---------\n");
                        printf("Write SET again (llopen)\n");
                              printf("Flag: 0x%02X | Address: 0x%02X |
Control: 0x%02X | BCC: 0x%02X | Flag: 0x%02X\n", bufW[0], bufW[1],
bufW[2], bufW[3], bufW[4]);
                                int bytesW = writeBytesSerialPort(bufW,
BUF_SIZE);
                             printf("%d bytes written (SET)(llopen)\n",
bytesW);

printf("-----------------------------------------------------------------
---------\n");
                        frameCounter++;
                        retransmitionsCount++;
                    }

                    alarm(connectionParameters.timeout);
                    alarmEnabled = TRUE;

                }

                unsigned char byte;

                if(readByteSerialPort(&byte) > 0){
                    switch (stateW) {
                        case START_STATE: {
                            if(byte == FLAG) {
                                bufW2[0] = byte;
                                stateW = FLAG_STATE;
                            }
                            break;}
                        case FLAG_STATE:
                            {if(byte == FLAG) {
                                bufW2[0] = byte;
                            } else if (byte == ADDRESS_RECEIVE) {
                                bufW2[1] = byte;
```

```
                                a_prov1 = byte;
                                stateW = A_STATE;
                            } else {
                                stateW = START_STATE;
                            }
                            break;}
                        case A_STATE:
                            {if(byte == FLAG) {
                                bufW2[0] = byte;
                                stateW = FLAG_STATE;
                            } else if (byte == CONTROL_UA) {
                                bufW2[2] = byte;
                                c_prov1 = byte;
                                stateW = C_STATE;
                            } else {
                                stateW = START_STATE;
                            }
                            break;}
                        case C_STATE:
                            {if(byte == FLAG) {
                                bufW2[0] = byte;
                                stateW = FLAG_STATE;
                            } else if (byte == (a_prov1 ^ c_prov1)) {
                                bufW2[3] = byte;
                                stateW = BCC_STATE;
                            } else {
                                stateW = START_STATE;
                            }
                            break;}
                        case BCC_STATE:
                            {if(byte == FLAG) {
                                bufW2[4] = byte;
                                stateW = STOP_STATE;
                                frameCounterReceived++;
                                retransmitionsCount = 0;
                                alarm(0);
                            } else {
                                stateW = START_STATE;
                            }
                            break;}
                        default:
                            {break;}
                    }
                }
            }


printf("-------------------------------------------------------------------
---------\n");
            printf("Collected from Read (llopen): ");
            for (int i = 0; i < 5; i++){
                printf("0x%02X |", bufW2[i]);
            }
            printf("\n");

printf("-------------------------------------------------------------------
---------\n");
```

```
            alarmEnabled = FALSE;
            alarmCount = 0;
            if (stateW != STOP_STATE) { return -1;}

            break;}

        case LlRx:
            {
            // READ WAITS FOR SET
             unsigned char bufR[BUF_SIZE + 1] = {0}; // +1: Save space
for the final '\0' char
            state_t stateR = START_STATE;
            int a_prov2 = 0;
            int c_prov2 = 0;

            while(stateR != STOP_STATE) {
                unsigned char byte;
                if(readByteSerialPort(&byte)){
                    byteCount++;
                    switch (stateR) {
                        case START_STATE:
                            {if(byte == FLAG) {
                                bufR[0] = byte;
                                stateR = FLAG_STATE;
                            }
                            break;}
                        case FLAG_STATE:
                            {if(byte == FLAG) {
                                bufR[0] = byte;
                            } else if (byte == ADDRESS_SEND) {
                                bufR[1] = byte;
                                a_prov2 = byte;
                                stateR = A_STATE;
                            } else {
                                stateR = START_STATE;
                            }
                            break;}
                        case A_STATE:
                            {if(byte == FLAG) {
                                bufR[0] = byte;
                                stateR = FLAG_STATE;
                            } else if (byte == CONTROL_SET) {
                                bufR[2] = byte;
                                c_prov2 = byte;
                                stateR = C_STATE;
                            } else {
                                stateR = START_STATE;
                            }
                            break;}
                        case C_STATE:
                            {if(byte == FLAG) {
                                bufR[0] = byte;
                                stateR = FLAG_STATE;
                            } else if (byte == (a_prov2 ^ c_prov2)) {
                                bufR[3] = byte;
                                stateR = BCC_STATE;
                            } else {
                                stateR = START_STATE;
```

```
                            }
                            break;}
                        case BCC_STATE:
                            {if(byte == FLAG) {
                                bufR[4] = byte;
                                stateR = STOP_STATE;
                                frameCounterReceived++;
                            } else {
                                stateR = START_STATE;
                            }
                            break;}
                        default:
                            {break;}
                        }
                    }

                }


printf("-----------------------------------------------------------------
---------\n");
                printf("Collected from Write (llopen) : ");
                for (int i = 0; i < 5; i++){
                    printf("0x%02X |", bufR[i]);
                }
                printf("\n");

printf("-----------------------------------------------------------------
---------\n");

                //READ RESPONDE DE VOLTA
                unsigned char bufR2[BUF_SIZE];
                unsigned char BCC1R = ADDRESS_RECEIVE ^ CONTROL_UA;
                bufR2[0] = FLAG;
                bufR2[1] = ADDRESS_RECEIVE; //0X01
                bufR2[2] = CONTROL_UA; //0X07
                bufR2[3] = BCC1R;
                bufR2[4] = FLAG;

printf("-----------------------------------------------------------------
---------\n");
                printf("Read UA (llopen) \n");
                 printf("Flag: 0x%02X | Address: 0x%02X | Control: 0x%02X |
BCC: 0x%02X | Flag: 0x%02X\n", bufR2[0], bufR2[1], bufR2[2], bufR2[3],
bufR2[4]);
                int bytesR = writeBytesSerialPort(bufR2, BUF_SIZE);
                printf("%d bytes written (UA)(llopen)\n", bytesR);

printf("-----------------------------------------------------------------
---------\n");
                frameCounter++;

                break;}

            default:{
                printf("ERROR: role unknown");
                return -1;}
        }
```

```
    return fd;

}



////////////////////////////////////////////////
// LLWRITE
////////////////////////////////////////////////
int llwrite(const unsigned char *buf, int bufSize) {

     unsigned int sizeOfFrame = bufSize + 6; // 6 = 2*FLAG + ADDRESS +
CONTROL + BCC1 + BCC2
    unsigned char *frame = (unsigned char *) malloc(sizeOfFrame);
    frame[0] = FLAG; // Flag
    frame[1] = ADDRESS_SEND; // A = 0X03

    if (Ns_t == 0){
        frame[2] = CONTROL_I_N0; // C = 0x00
    } else if (Ns_t == 1) {
        frame[2] = CONTROL_I_N1; // C = 0x80
    } else {
        printf("ERROR: INFO FRAME C\n");
        return -1;
    }

    frame[3] = frame[1] ^ frame[2]; // A^C

    memcpy (frame+4, buf, bufSize); // D1...Dn

    int k = 4;
    for(int i = 0; i < bufSize; i++){
        if(buf[i] == FLAG){ // If 0x7e -> 0x7d 0x5e
            frame = realloc(frame,++sizeOfFrame);
            frame[k] = ESCAPE;
            k++;
            frame[k] = XOR_FLAG;
            k++;
        } else if (buf[i] == ESCAPE) {  // If 0x7d -> 0x7d 0x5dc
            frame = realloc(frame,++sizeOfFrame);
            frame[k] = ESCAPE;
            k++;
            frame[k] = XOR_ESCAPE;
            k++;
        } else {
            frame[k] = buf[i];
            k++;
        }
    }

    unsigned char BCC2 = buf[0];
      for (unsigned int i = 1 ; i < bufSize ; i++) { BCC2 = BCC2 ^
buf[i]; } // BCC2 = D1^D2^...^Dn

    if(BCC2 == FLAG){ // If 0x7e -> 0x7d 0x5e
        frame = realloc(frame,++sizeOfFrame);
        frame[k] = ESCAPE;
        k++;
```

```
        frame[k] = XOR_FLAG;
        k++;
    } else if (BCC2 == ESCAPE) {  // If 0x7d -> 0x7d 0x5dc
        frame = realloc(frame,++sizeOfFrame);
        frame[k] = ESCAPE;
        k++;
        frame[k] = XOR_ESCAPE;
        k++;
    } else {
        frame[k] = BCC2;
        k++;
    }

    frame[k] = FLAG; // Flag
    k++;


printf("-----------------------------------------------------------------
---------\n");
    printf("Write I (llwrite)\n");
    int bytesW = writeBytesSerialPort(frame, k);
    printf("%d bytes written (llwrite)\n", bytesW);

printf("-----------------------------------------------------------------
---------\n");
    frameCounter++;

    //WRITE RECEBE DE VOLTA
    (void)signal(SIGALRM, alarmHandler);
    alarmEnabled = FALSE;
    alarmCount = 0;
     unsigned char bufllwrite[BUF_SIZE + 1] = {0}; // +1: Save space for
the final '\0' char
    state_t statellwrite = START_STATE;
    int a_prov_llwrite = 0;
    int c_prov_llwrite = 0;


         while(statellwrite  !=  STOP_STATE  &&  retransmitionsCount  <
connectionParametersCopy.nRetransmissions) {
        if (alarmEnabled == FALSE) {
            if(alarmCount != 0){

printf("-----------------------------------------------------------------
---------\n");
                printf("Write I again (llwrite)\n");
                int bytesW = writeBytesSerialPort(frame, k);
                printf("%d bytes written (llwrite)\n", bytesW);

printf("-----------------------------------------------------------------
---------\n");
                frameCounter++;
                retransmitionsCount++;
            }

            alarm(connectionParametersCopy.timeout);
            alarmEnabled = TRUE;
        }
```

```
        unsigned char byte;
        if(readByteSerialPort(&byte) > 0){
            switch (statellwrite) {
                case START_STATE:
                    {if(byte == FLAG) {
                        bufllwrite[0] = byte;
                        statellwrite = FLAG_STATE;
                    }
                    break;}
                case FLAG_STATE:
                    {if(byte == FLAG) {
                        bufllwrite[0] = byte;
                    } else if (byte == ADDRESS_RECEIVE) {
                        bufllwrite[1] = byte;
                        a_prov_llwrite = byte;
                        statellwrite = A_STATE;
                    } else {
                        statellwrite = START_STATE;
                    }
                    break;}
                case A_STATE:
                    {if(byte == FLAG) {
                        bufllwrite[0] = byte;
                        statellwrite = FLAG_STATE;
                     } else if (Ns_t == 0 && byte == CONTROL_RR1 && Nr_r
== 1) {
                        bufllwrite[2] = byte;
                        c_prov_llwrite = byte;
                        statellwrite = C_STATE;
                        Ns_t++;
                        Nr_r = 0;
                     } else if (Ns_t == 1 && byte == CONTROL_RR0 && Nr_r
== 0) {
                        bufllwrite[2] = byte;
                        c_prov_llwrite = byte;
                        statellwrite = C_STATE;
                        Ns_t = 0;
                        Nr_r++;
                       } else if (Ns_t == 0 && byte == CONTROL_REJ0 &&
Nr_r == 1) {
                        statellwrite = START_STATE;
                        alarmEnabled = FALSE;

printf("----------------------------------------------------------------
---------\n");
                                printf("Write I again (recived a REJ0)
(llwrite)\n");
                        int bytesW = writeBytesSerialPort(frame, k);
                        printf("%d bytes written (llwrite)\n", bytesW);

printf("----------------------------------------------------------------
---------\n");
                        frameCounter++;
                        alarm(0);
                        retransmitionsCount = 0;
                        alarmCount = 0;
```

```
                        } else if (Ns_t == 1 && byte == CONTROL_REJ1 &&
Nr_r == 0) {
                            statellwrite = START_STATE;
                            alarmEnabled = FALSE;

printf("-----------------------------------------------------------------
---------\n");
                                    printf("Write I again (recived a REJ1)
(llwrite)\n");
                            int bytesW = writeBytesSerialPort(frame, k);
                            printf("%d bytes written (llwrite)\n", bytesW);

printf("-----------------------------------------------------------------
---------\n");
                            frameCounter++;
                            alarm(0);
                            retransmitionsCount = 0;
                            alarmCount = 0;
                        } else {
                            statellwrite = START_STATE;
                        }
                        break;}
                    case C_STATE:
                        {if(byte == FLAG) {
                            bufllwrite[0] = byte;
                            statellwrite = FLAG_STATE;
                                    } else if (byte == (a_prov_llwrite ^
c_prov_llwrite)) {
                            bufllwrite[3] = byte;
                            statellwrite = BCC_STATE;
                        } else {
                           statellwrite = START_STATE;
                        }
                        break;}
                    case BCC_STATE:
                        {if(byte == FLAG) {
                            bufllwrite[4] = byte;
                            statellwrite = STOP_STATE;
                            frameCounterReceived++;
                            retransmitionsCount = 0;
                            alarm(0);
                        } else {
                            statellwrite = START_STATE;
                        }
                        break;}
                    default:
                        {break;}
                }
            }
        }


printf("-----------------------------------------------------------------
---------\n");
    printf("Collected from Read (llwrite): ");
    for (int i = 0; i < 5; i++){
        printf("0x%02X |", bufllwrite[i]);
    }
```

```
    printf("\n");

printf("-------------------------------------------------------------------
---------\n");

    free(frame);

    alarmEnabled = FALSE;
    alarmCount = 0;
    if(statellwrite == STOP_STATE) {return sizeOfFrame;}

    return -1;
}



////////////////////////////////////////////////
// LLREAD
////////////////////////////////////////////////
int llread(unsigned char *packet) {

    // READ RECEBE
    state_t stateR = START_STATE;
    int a_prov2 = 0;
    int c_prov2 = 0;
    int i = 0;

    while(stateR != STOP_STATE) {
        unsigned char byte;
        if(readByteSerialPort(&byte) > 0){
            byteCount++;
            switch (stateR) {
                case START_STATE:
                    {if(byte == FLAG) {
                        stateR = FLAG_STATE;
                    }
                    break;}
                case FLAG_STATE:
                    {if(byte == FLAG) {
                    } else if (byte == ADDRESS_SEND) {
                        a_prov2 = byte;
                        stateR = A_STATE;
                    } else if (byte == 0x00) {
                        // Ignore and stay
                    } else {
                        stateR = START_STATE;
                    }
                    break;}
                case A_STATE:
                    {if(byte == FLAG) {
                        stateR = FLAG_STATE;
                      } else if ((Ns_t == 0 && byte == CONTROL_I_N0) ||
(Ns_t == 1 && byte == CONTROL_I_N1)){
                        c_prov2 = byte;
                        stateR = C_STATE;
                        counterNotSets++;
                    } else if (byte == CONTROL_SET && counterNotSets ==
0){
```

```
                    resendUA();
                    stateR = START_STATE;
                } else if (byte == 0x00) {
                    // Ignore and stay
                } else {
                    stateR = START_STATE;
                }
                break;}
            case C_STATE:
                {if(byte == FLAG) {
                    stateR = FLAG_STATE;
                } else if (byte == (a_prov2 ^ c_prov2)) {
                    stateR = BCC_STATE;
                } else if (byte == 0x00) {
                    // Ignore and stay
                } else {
                    stateR = START_STATE;
                }
                break;}
            case BCC_STATE: // DESTUFF AND BCC CHECKING
                {if (byte == ESCAPE) {
                    stateR = DESTUFFING_STATE;
                } else if (byte == FLAG) {
                    unsigned int bcc2 = packet[--i];
                    packet[i] = '\0';
                    unsigned int bcc_calculo = packet[0];
                    for(int j = 1 ; j<i ; j++){
                        bcc_calculo = bcc_calculo ^ packet[j];
                    }

                    if(bcc2 == bcc_calculo) {
                        stateR = STOP_STATE;
                        frameCounterReceived++;
                        if (Nr_r == 0){ // Respond with RR0
                                    unsigned char frame[5] = {FLAG,
ADDRESS_RECEIVE, CONTROL_RR0, ADDRESS_RECEIVE ^ CONTROL_RR0, FLAG};
                                writeBytesSerialPort(frame, 5);

printf("-------------------------------------------------------------------
---------\n");
                                printf("Read RR0 (llread)\n");
                                 printf("Flag: 0x%02X | Address: 0x%02X
| Control: 0x%02X | BCC: 0x%02X | Flag: 0x%02X\n", frame[0], frame[1],
frame[2], frame[3], frame[4]);
                                                printf("%d bytes written
(RR0)(llread)\n", 5);

printf("-------------------------------------------------------------------
---------\n");
                                frameCounter++;
                                Nr_r = 1;
                                Ns_t = 0;
                                stateR = STOP_STATE;
                                            memcpy(lastFrame, frame,
sizeof(frame));
                            } else { // Respond with RR1
                                    unsigned char frame[5] = {FLAG,
ADDRESS_RECEIVE, CONTROL_RR1, ADDRESS_RECEIVE ^ CONTROL_RR1, FLAG};
```

```
                                writeBytesSerialPort(frame, 5);

printf("-----------------------------------------------------------------
---------\n");
                                printf("Read RR1 (llread)\n");
                                  printf("Flag: 0x%02X | Address: 0x%02X
| Control: 0x%02X | BCC: 0x%02X | Flag: 0x%02X\n", frame[0], frame[1],
frame[2], frame[3], frame[4]);
                                                    printf("%d bytes written
(RR1)(llread)\n", 5);

printf("-----------------------------------------------------------------
---------\n");
                                frameCounter++;
                                Nr_r = 0;
                                Ns_t = 1;
                                stateR = STOP_STATE;
                                                    memcpy(lastFrame, frame,
sizeof(frame));

                            }
                            return i;
                        } else {
                            if (Nr_r == 0){ // Reject with REJ0
                                                unsigned char frame[5] = {FLAG,
ADDRESS_RECEIVE, CONTROL_REJ0, ADDRESS_RECEIVE ^ CONTROL_REJ0, FLAG};
                                writeBytesSerialPort(frame, 5);

printf("-----------------------------------------------------------------
---------\n");
                                printf("Read REJ0 (llread)\n");
                                  printf("Flag: 0x%02X | Address: 0x%02X
| Control: 0x%02X | BCC: 0x%02X | Flag: 0x%02X\n", frame[0], frame[1],
frame[2], frame[3], frame[4]);
                                                    printf("%d bytes written
(REJ0)(llread)\n", 5);

printf("-----------------------------------------------------------------
---------\n");
                                frameCounter++;
                                stateR = START_STATE;
                                bcc2 = 0;
                                memset(packet, 0, i);
                                i = 0;
                            } else { // Reject with REJ1
                                                unsigned char frame[5] = {FLAG,
ADDRESS_RECEIVE, CONTROL_REJ1, ADDRESS_RECEIVE ^ CONTROL_REJ1, FLAG};
                                writeBytesSerialPort(frame, 5);

printf("-----------------------------------------------------------------
---------\n");
                                printf("Read REJ1 (llread)\n");
                                  printf("Flag: 0x%02X | Address: 0x%02X
| Control: 0x%02X | BCC: 0x%02X | Flag: 0x%02X\n", frame[0], frame[1],
frame[2], frame[3], frame[4]);
                                                    printf("%d bytes written
(REJ1)(llread)\n", 5);
```

```
printf("-------------------------------------------------------------
---------\n");
                                frameCounter++;
                                stateR = START_STATE;
                                bcc2 = 0;
                                memset(packet, 0, i);
                                i = 0;
                            }
                        }

                    } else {
                        packet[i++] = byte;
                    }
                    break;}
                case DESTUFFING_STATE:
                    {if(byte == XOR_ESCAPE){
                        packet[i++] = ESCAPE;
                        stateR = BCC_STATE;
                    } else if (byte == XOR_FLAG) {
                        packet[i++] = FLAG;
                        stateR = BCC_STATE;
                    } else {
                        stateR = STOP_STATE;
                        return -1;
                    }

                    break;}
                default:
                    {break;}
                }
        }

    }
    return -1;
}



////////////////////////////////////////////////
// LLCLOSE
////////////////////////////////////////////////
int llclose(int showStatistics) {

    switch (connectionParametersCopy.role)
    {
    case LlTx:
        {
        unsigned char bufW[BUF_SIZE];

        unsigned char BCC1W = ADDRESS_SEND ^ CONTROL_DISC;
        bufW[0] = FLAG;
        bufW[1] = ADDRESS_SEND; //0X03
        bufW[2] = CONTROL_DISC; //0x0B
        bufW[3] = BCC1W;
        bufW[4] = FLAG;
```

```
printf("------------------------------------------------------------------
---------\n");
        printf("Write DISC (llclose)\n");
         printf("Flag: 0x%02X | Address: 0x%02X | Control: 0x%02X | BCC:
0x%02X | Flag: 0x%02X\n", bufW[0], bufW[1], bufW[2], bufW[3], bufW[4]);
        int bytesW = writeBytesSerialPort(bufW, BUF_SIZE);
        printf("%d bytes written (DISC)(llclose)\n", bytesW);

printf("------------------------------------------------------------------
---------\n");
        frameCounter++;

        //WRITE RECEBE DE VOLTA
        (void)signal(SIGALRM, alarmHandler);
        alarmEnabled = FALSE;
        alarmCount = 0;
         unsigned char bufW2[BUF_SIZE + 1] = {0}; // +1: Save space for
the final '\0' char
        state_t stateW = START_STATE;
        int a_prov1 = 0;
        int c_prov1 = 0;

                while(stateW != STOP_STATE && retransmitionsCount <
connectionParametersCopy.nRetransmissions) {

            if (alarmEnabled == FALSE) {
                if(alarmCount != 0){
                    unsigned char bufW[BUF_SIZE];

                    unsigned char BCC1W = ADDRESS_SEND ^ CONTROL_DISC;
                    bufW[0] = FLAG;
                    bufW[1] = ADDRESS_SEND; //0X03
                    bufW[2] = CONTROL_DISC; //0x0B
                    bufW[3] = BCC1W;
                    bufW[4] = FLAG;

printf("------------------------------------------------------------------
---------\n");
                    printf("Write DISC again (llclose)\n");
                      printf("Flag: 0x%02X | Address: 0x%02X | Control:
0x%02X | BCC: 0x%02X | Flag: 0x%02X\n", bufW[0], bufW[1], bufW[2],
bufW[3], bufW[4]);
                    int bytesW = writeBytesSerialPort(bufW, BUF_SIZE);
                           printf("%d bytes written (DISC)(llclose)\n",
bytesW);

printf("------------------------------------------------------------------
---------\n");
                    frameCounter++;
                    retransmitionsCount++;
                }

                alarm(connectionParametersCopy.timeout);
                alarmEnabled = TRUE;

            }
```

```
            unsigned char byte;

            if(readByteSerialPort(&byte) > 0){
                switch (stateW) {
                    case START_STATE:
                        {if(byte == FLAG) {
                            bufW2[0] = byte;
                            stateW = FLAG_STATE;
                        }
                        break;}
                    case FLAG_STATE:
                        {if(byte == FLAG) {
                            bufW2[0] = byte;
                        } else if (byte == ADDRESS_RECEIVE) {
                            bufW2[1] = byte;
                            a_prov1 = byte;
                            stateW = A_STATE;
                        } else {
                            stateW = START_STATE;
                        }
                        break;}
                    case A_STATE:
                        {if(byte == FLAG) {
                            bufW2[0] = byte;
                            stateW = FLAG_STATE;
                        } else if (byte == CONTROL_DISC) {
                            bufW2[2] = byte;
                            c_prov1 = byte;
                            stateW = C_STATE;
                        } else {
                            stateW = START_STATE;
                        }
                        break;}
                    case C_STATE:
                        {if(byte == FLAG) {
                            bufW2[0] = byte;
                            stateW = FLAG_STATE;
                        } else if (byte == (a_prov1 ^ c_prov1)) {
                            bufW2[3] = byte;
                            stateW = BCC_STATE;
                        } else {
                            stateW = START_STATE;
                        }
                        break;}
                    case BCC_STATE:
                        {if(byte == FLAG) {
                            bufW2[4] = byte;
                            stateW = STOP_STATE;
                            frameCounterReceived++;
                            alarm(0);


printf("------------------------------------------------------------------
---------\n");
                            printf("Collected from Read (llclose): ");
                            for (int i = 0; i < 5; i++){
                                printf("0x%02X |", bufW2[i]);
                            }
```

```
                            printf("\n");

printf("------------------------------------------------------------------
---------\n");

                            unsigned char bufAU[BUF_SIZE];
                                  unsigned char BCC1AU = ADDRESS_SEND ^
CONTROL_UA;
                            bufAU[0] = FLAG;
                            bufAU[1] = ADDRESS_SEND; //0X03
                            bufAU[2] = CONTROL_UA; //0X07
                            bufAU[3] = BCC1AU;
                            bufAU[4] = FLAG;

printf("------------------------------------------------------------------
---------\n");
                            printf("Write UA (llclose)\n");
                               printf("Flag: 0x%02X | Address: 0x%02X |
Control: 0x%02X | BCC: 0x%02X | Flag: 0x%02X\n", bufW[0], bufW[1],
bufW[2], bufW[3], bufW[4]);
                                 int bytesAU = writeBytesSerialPort(bufAU,
BUF_SIZE);
                                printf("%d bytes written (UA)(llclose)\n",
bytesAU);

printf("------------------------------------------------------------------
---------\n");
                            frameCounter++;
                            retransmitionsCount = 0;

                        } else {
                            stateW = START_STATE;
                        }
                        break;}
                    default:
                        {break;}
                }
            }
        }


        alarmEnabled = FALSE;
        alarmCount = 0;
        if (stateW != STOP_STATE) { return -1;}

        break;}
    case LlRx:
        {
        // READ RECEBE
         unsigned char bufR[BUF_SIZE + 1] = {0}; // +1: Save space for
the final '\0' char
        state_t stateR = START_STATE;
        int a_prov2 = 0;
        int c_prov2 = 0;

        while(stateR != STOP_STATE) {
            unsigned char byte;
            if(readByteSerialPort(&byte) > 0){
```

```
                byteCount++;
                switch (stateR) {
                    case START_STATE:
                        {if(byte == FLAG) {
                            bufR[0] = byte;
                            stateR = FLAG_STATE;
                        }
                        break;}
                    case FLAG_STATE:
                        {if(byte == FLAG) {
                            bufR[0] = byte;
                        } else if (byte == ADDRESS_SEND) {
                            bufR[1] = byte;
                            a_prov2 = byte;
                            stateR = A_STATE;
                        } else {
                            stateR = START_STATE;
                        }
                        break;}
                    case A_STATE:
                        {if(byte == FLAG) {
                            bufR[0] = byte;
                            stateR = FLAG_STATE;
                        } else if (byte == CONTROL_DISC) {
                            bufR[2] = byte;
                            c_prov2 = byte;
                            stateR = C_STATE;
                             } else if (byte == CONTROL_I_N0 || byte ==
CONTROL_I_N1) {
                            writeBytesSerialPort(lastFrame,5);
                        } else {
                            stateR = START_STATE;
                        }
                        break;}
                    case C_STATE:
                        {if(byte == FLAG) {
                            bufR[0] = byte;
                            stateR = FLAG_STATE;
                        } else if (byte == (a_prov2 ^ c_prov2)) {
                            bufR[3] = byte;
                            stateR = BCC_STATE;
                        } else {
                            stateR = START_STATE;
                        }
                        break;}
                    case BCC_STATE:
                        {if(byte == FLAG) {
                            bufR[4] = byte;
                            stateR = STOP_STATE;
                            frameCounterReceived++;
                        } else {
                            stateR = START_STATE;
                        }
                        break;}
                    default:
                        {break;}
                    }
            }
```

```
        }


printf("-----------------------------------------------------------------
---------\n");
        printf("Collected from Write (llclose): ");
        for (int i = 0; i < 5; i++){
            printf("0x%02X |", bufR[i]);
        }
        printf("\n");

printf("-----------------------------------------------------------------
---------\n");

        //READ RESPONDE DE VOLTA
        unsigned char bufR2[BUF_SIZE];
        unsigned char BCC1R = ADDRESS_RECEIVE ^ CONTROL_DISC;
        bufR2[0] = FLAG;
        bufR2[1] = ADDRESS_RECEIVE; //0X01
        bufR2[2] = CONTROL_DISC; //0X0B
        bufR2[3] = BCC1R;
        bufR2[4] = FLAG;

printf("-----------------------------------------------------------------
---------\n");
        printf("Read DISC (llclose): \n");
         printf("Flag: 0x%02X | Address: 0x%02X | Control: 0x%02X | BCC:
0x%02X  |  Flag:  0x%02X\n",  bufR2[0],  bufR2[1],  bufR2[2],  bufR2[3],
bufR2[4]);
        int bytesR = writeBytesSerialPort(bufR2, BUF_SIZE);
        printf("%d bytes written (DISC)(llclose)\n", bytesR);

printf("-----------------------------------------------------------------
---------\n");
        frameCounter++;

        stateR = START_STATE;
        a_prov2 = 0;
        c_prov2 = 0;

        while(stateR != STOP_STATE) {
            unsigned char byte;
            if(readByteSerialPort(&byte)){
                byteCount++;
                switch (stateR) {
                    case START_STATE:
                        {if(byte == FLAG) {
                            bufR[0] = byte;
                            stateR = FLAG_STATE;
                        }
                        break;}
                    case FLAG_STATE:
                        {if(byte == FLAG) {
                            bufR[0] = byte;
                        } else if (byte == ADDRESS_SEND) {
                            bufR[1] = byte;
                            a_prov2 = byte;
```

```
                            stateR = A_STATE;
                        } else {
                            stateR = START_STATE;
                        }
                        break;}
                    case A_STATE:
                        {if(byte == FLAG) {
                            bufR[0] = byte;
                            stateR = FLAG_STATE;
                        } else if (byte == CONTROL_UA) {
                            bufR[2] = byte;
                            c_prov2 = byte;
                            stateR = C_STATE;
                        } else {
                            stateR = START_STATE;
                        }
                        break;}
                    case C_STATE:
                        {if(byte == FLAG) {
                            bufR[0] = byte;
                            stateR = FLAG_STATE;
                        } else if (byte == (a_prov2 ^ c_prov2)) {
                            bufR[3] = byte;
                            stateR = BCC_STATE;
                        } else {
                            stateR = START_STATE;
                        }
                        break;}
                    case BCC_STATE:
                        {if(byte == FLAG) {
                            bufR[4] = byte;
                            stateR = STOP_STATE;
                            frameCounterReceived++;
                        } else {
                            stateR = START_STATE;
                        }
                        break;}
                    default:
                        {break;}
                    }
                }

            }

printf("-----------------------------------------------------------------
---------\n");
            printf("Collected from Write (llclose): ");
            for (int i = 0; i < 5; i++){
                printf("0x%02X |", bufR[i]);
            }
            printf("\n");

printf("-----------------------------------------------------------------
---------\n");

            break;
           }
```

```
    default:
        {printf("error: role unknown");

        return -1;}
    }

    endTotal = clock();
    time_used = ((double)(end - start))/CLOCKS_PER_SEC;
    total_time_used = ((double)(endTotal - startTotal))/CLOCKS_PER_SEC;

    if(showStatistics == TRUE) {
        printf("\n");
        printf("\n");

printf("-----------------------------------------------------------------
---------\n");
        printf("STATISTICS: \n");
        printf("\n");
        switch (connectionParametersCopy.role)
        {
        case LlTx:
                        {printf("Transmitter sent %u frames in this file
transfer.\n", frameCounter);
                    printf("Transmitter collected %u frames in this file
transfer.\n", frameCounterReceived);
            printf("\n");
               printf("Transmitter took %f seconds to execute the whole
program.\n", total_time_used);
              printf("Transmitter took %f seconds to transmit the file.
\n", time_used);
            printf("\n");
                                printf("Transmitter made %u alarms.\n",
alarmTotalAlarmCount);
            break;}
        case LlRx:
            {
            int bitCount = byteCount*8;
            double bitPerSec = bitCount/total_time_used;
            double bytePerSec = byteCount/total_time_used;
             printf("Receiver sent %u frames in this file transfer.\n",
frameCounter);
                    printf("Receiver collected %u frames in this file
transfer.\n", frameCounterReceived);
            printf("\n");
                printf("Receiver took %f seconds to execute the whole
program.\n", total_time_used);
             printf("Receiver took %f seconds to receive the file. \n",
time_used);
            printf("\n");
                printf("Receiver's rate of received data: %f bit/s (%f
bytes/s). \n", bitPerSec, bytePerSec);
            break;}

        default:
            {break;}
        }
```

```
printf("------------------------------------------------------------
---------\n");
    }

    int clstat = closeSerialPort();
    return clstat;
}
```

application_layer.c

```
// Application layer protocol implementation

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

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <fenv.h>
#include <time.h>

extern clock_t startTotal, endTotal;
extern clock_t start, end;



unsigned char * createControlPacket(const char* filename, int fileSize,
unsigned int* controlPacketSize, const unsigned int cField) {

    int L1 = 0;
    unsigned long tempSize = fileSize;
    while (tempSize > 0) {
        L1++;
        tempSize >>= 8;
    }

     const int L2 = strlen(filename); // Number of bytes needed to store
the file name

    *controlPacketSize = 1+2+L1+2+L2; // Control packet total size

    unsigned char *packet = (unsigned char*)malloc(*controlPacketSize);
      packet[0] = cField; // Set the 'cField' field of the control
packet. (values:  1 -> start or 3 -> end)
     packet[1] = 0; // Marking that we are going to place info of file
size. (T1)
    packet[2] = L1; // Set the size of the fileSize content.


    for (unsigned char i = 0; i < L1; i++) {
              packet[2+L1-i] = fileSize & 0xFF;   // Write the least
significant byte of length.
```

```
        fileSize>>=8;                       // Shift length to the right
by 8 bits.
    }

    int packetPosition = 2 + L1;
     packet[packetPosition] = 1; // Marking that we are going to place
info of file name. (T2)
    packetPosition++;
      packet[packetPosition] = L2; // Set the size of the filename
content.
    packetPosition++;

     memcpy(packet+packetPosition, filename, L2); // Copy the filename
into the packet.
    return packet;
}



unsigned char * createDataPacket(unsigned char* payload, int
payloadSize, unsigned int* dataPacketSize, unsigned char
sequenceNumber) {

    *dataPacketSize = 1+1+2+payloadSize; // Data packet total size

    unsigned char *packet = (unsigned char*)malloc(*dataPacketSize);
    packet[0] = 2; // Control field (value: 2 -> data)
     packet[1] = sequenceNumber; // Set the sequenceNumber field of the
data packet.
    packet[2] = payloadSize / 256; // Set L2.
    packet[3] = payloadSize % 256; // Set L1.

      memcpy(packet+4, payload, payloadSize); // Copy the payload into
the packet.
    return packet;
}



void analyseControlPacket(unsigned char* packet, int sizePacket,
unsigned char* receivedFilename) {

    unsigned char receivedFilesize;
    unsigned char L1 = packet [2]; // Size of fileSize content.
     unsigned char tempAux[L1]; // Auxiliar array to store the fileSize
content.
      memcpy(tempAux, packet+3, L1); // Copy the fileSize content into
tempAux.

    for (unsigned char i = 0; i < L1; i++) {
        receivedFilesize |= (tempAux[L1-i-1] << (8*i));
    }

    unsigned char L2 = packet[3+L1+1]; // Size of filename contents.
    receivedFilename = (unsigned char*) malloc (L2);
    memcpy(receivedFilename, packet+3+L1+1+1, L2);
}
```

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

    switch (role[0]) {
        case 't':
            {connectionParameters.role=LlTx;
            break;}
        case 'r':
            {connectionParameters.role=LlRx;
            break;}
        default:
            {printf("ERROR: unknown user.role");
            return;}
    }

    strcpy(connectionParameters.serialPort, serialPort);
    connectionParameters.baudRate = baudRate;
    connectionParameters.nRetransmissions=nTries;
    connectionParameters.timeout=timeout;

    if (llopen(connectionParameters) == -1) {
        printf("ERROR: llopen failed");
        return;
    }

    start = clock();
    switch (connectionParameters.role) {
        case LlTx:
        {
             FILE* file = fopen(filename, "rb"); // Open a binary file
for reading. (The file must exist.)
            if (file == NULL) {
                perror("ERROR: file not found\n");
                return;
            }
            int prev = ftell(file); // Store current position.
            fseek(file,0L,SEEK_END); // Go to end of file.
            long int fileSize = ftell(file) - prev; // Get file size.
            fseek(file,prev,SEEK_SET); // Go back to previous position.


            unsigned int startingCPSize; // Control packet size
                            unsigned  char  *startingControlPacket  =
createControlPacket(filename,  fileSize,  &startingCPSize,  1);  //  Get
start control packet.

            // Send data in buf with size bufSize.
            if (llwrite(startingControlPacket, startingCPSize) == -1) {
                        printf("ERROR: llwrite starting control packet
failed");
                return;
            }
```

```
            unsigned char sequenceNum = 0;
            long int bytesRemaining = fileSize;
                  unsigned char* content = (unsigned char* ) malloc
(sizeof(unsigned char) * fileSize); // Allocate memory for file
content.
               fread(content, 1, fileSize, file); // Read file content
into buffer.

            while (bytesRemaining > 0) {

                int payloadSize=0;
                if (bytesRemaining > (long int) MAX_PAYLOAD_SIZE){
                    payloadSize = MAX_PAYLOAD_SIZE;
                } else {
                    payloadSize = bytesRemaining;
                }

                           unsigned char* payload = (unsigned char* )
malloc(payloadSize); // Allocate memory for payload.
                 memcpy(payload, content, payloadSize); // Copy payload
from content buffer.

                unsigned int dataPacketSize; // Data packet size.
                  unsigned char *dataPacket = createDataPacket(payload,
payloadSize, &dataPacketSize, sequenceNum);

                if (llwrite(dataPacket, dataPacketSize) == -1) {
                    printf("ERROR: llwrite data packet failed");
                    return;
                }

                sequenceNum = (sequenceNum+1) % 255;

                bytesRemaining = bytesRemaining - payloadSize;
                content += payloadSize;
            }

            unsigned int endingCPSize; // Control packet size.
                               unsigned char *endingControlPacket =
createControlPacket(filename, fileSize, &endingCPSize, 3); // Get start
control packet.
            if (llwrite(endingControlPacket, endingCPSize) == -1) {
                printf("ERROR: llwrite ending failed");
                return;
            }
            break;}

        case LlRx:
            {
                          unsigned char* packet = (unsigned char* )
malloc(MAX_PAYLOAD_SIZE); // Allocate memory for packet.

            int sizePacket = -1;
            while (sizePacket == -1) {
                sizePacket = llread(packet);
            }

            unsigned char* receivedFilename;
```

```
            analyseControlPacket(packet, sizePacket, receivedFilename);

               FILE* fileReceived = fopen((char*) filename, "wb+"); //
Open an empty binary file for both reading and writing. (If file exists
its contents are destroyed)
             if (fileReceived==NULL) printf("ERROR: fileReceived failed
to open\n");


            while(TRUE) {
                int sizePacket2 = -1;
                while (sizePacket2 == -1) {
                    sizePacket2 = llread(packet);
                }

                int L1 = packet[3];
                int L2 = packet[2];
                int K = (256*L2) + L1;

                if(sizePacket2 == 0) {break;}
                else if (packet[0] == 3) { // An end control packet
                    break;
                }
                    else if (packet[0] != 3) {   // Not an end control
packet
                         unsigned char *bufReceived = (unsigned char* )
malloc (K);
                    memcpy(bufReceived,packet+4,K);
                          fwrite(bufReceived, sizeof(unsigned char), K,
fileReceived);
                    free(bufReceived);
                }
            }

            fclose(fileReceived);

            break;}


        default:
            {printf("ERROR: role unknown");
            return;}
    }
    end = clock();

    if(llclose(TRUE) == -1) {
        printf("ERROR: llclose failed");
        return;
    }

}
```