Data Link Protocol - 1st Lab

Class 11 Group 11

L.EIC 24/25

Redes de Computadores

Pedro Santos up202205900@fe.up.pt

Rodrigo de Sousa up202205751@fe.up.pt

Summary

The first lab work for RCOM focuses on the design and implementation of a Data Link Protocol based on the RS-232 serial communication standard, using the Stop-and-Wait strategy learned in class. This project required us to construct modular, layered code that could handle both data transmission and error control. While the serial_port.c interface, provided by the professors, manages low-level serial communication with functions to open, configure, and close the serial port, our work centred on designing the data_link.c and application_layer.c files. These files form the core of our protocol implementation and are critical to enabling reliable data transfer.

In data_link.c, we implemented the Stop-and-Wait protocol, which ensures that each frame transmitted receives an acknowledgment before the next one is sent. This file is responsible for framing, error detection, and retransmission management, which together provide the foundation for reliable communication. The application_layer.c module then builds on this by handling the higher-level logic required for sending and receiving complete data files. This layered approach was essential for managing complexity, as each module has a clearly defined role, allowing us to isolate responsibilities, debug more effectively, and understand how modular protocol design improves overall system reliability and maintainability.

Introduction

The aim of this work is to create a Data Link Protocol over RS-232, ensuring data integrity through Stop-and-Wait methodology. The report details the design, implementation, and testing phases of the protocol, illustrating each functional component's role.

- **Architecture**: Description of functional blocks and interfaces.
- **Code Structure**: Breakdown of APIs, data structures, and main functions.
- **Main Use Cases**: Explanation of key scenarios and function calls.
- **Logical Link Protocol**: Discussion on core functionalities with code excerpts.
- **Application Protocol**: High-level functionalities with code snippets.
- **Validation**: Test descriptions and performance results.
- **Protocol Efficiency**: Statistical evaluation comparing theoretical and practical outcomes.
- **Conclusions**: Summary of findings and lessons learned.

Architecture

The architecture of the Data Link Protocol project was designed with a modular approach, using a three-layer structure that promotes clarity, separation of responsibilities, and ease of testing and debugging. The three main layers: Application Layer, Link Layer, and Serial Layer; each serve distinct purposes in enabling reliable data transmission between the Transmitter and Receiver.

Application Layer: The Application Layer represents the top layer of our architecture, managing high-level interactions with the data being sent or received. This layer is responsible for coordinating file transfers by reading data from files and initiating data transmission to the lower layer (the Link Layer). It also handles file storage when data is received, providing an interface for users or programs to interact with files directly. By abstracting file operations, the Application Layer isolates the core data handling logic from the lower-level communication mechanics.

Link Layer: The Link Layer lies at the heart of the project and serves as an intermediary between the Application Layer and the Serial Layer. This layer implements the Stop-and-Wait protocol, ensuring data integrity through reliable frame transmission and acknowledgment handling. It is responsible for creating frames with appropriate headers, error-checking information, and data payloads. The Link Layer detects and handles frame loss or errors by managing retransmissions, leveraging a timeout mechanism to ensure the correct sequencing and receipt of frames. This layer is pivotal for maintaining reliable communication and separating data-link-specific logic from both high-level application concerns and low-level serial communication details.

Serial Layer: The Serial Layer is the bottom layer in the architecture and directly manages the physical communication over the RS-232 serial port. This layer, implemented in serial_port.c, was provided by the professors and is responsible for opening, configuring, and closing the serial port connection, as well as for reading and writing bytes. It is the direct interface between the Transmitter and Receiver, allowing the Link Layer to focus on protocol logic without needing to manage the specifics of serial communication. By abstracting these low-level operations, the Serial Layer provides a stable foundation upon which the higher layers can reliably function.

The Interface of the program depends if it is being run in two separate computers or not. If that is the case then the computer with the file is the transmitter and runs the code as the transmitter on the terminal and the receiver the code for the receiver also on the terminal. If not, for testing purposes, a user can open three terminals. One as the cable protocol, another as the transmitter and the last terminal as the receiver.

As the program runs it will print out messages so that the user can check how the file transfer is going, printing success and/or error messages, as well as a closing message.

Code Structure

Application Layer

In this layer, the only function that was created was the main applicationLayer that proceeds to call the Link Layer functions.

```
// Application layer main function.
void applicationLayer(const char *serialPort, const char *role, int baudRate,
int nTries, int timeout, const char *filename);
```

Link Layer

This layer contains all the functions that are useful for managing the connection between the application and the serial cable, as well as data structures.

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

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

```
typedef enum
{
    START,
    STOP,
    FLAG_SM,
    A_SM,
    C_SM,
    BCC_SM,
    READ_SM,
    ESC_SM
} LinkLayerSM;
```

```
//Handles the Alarms
void alarmHandler(int signal);

//Waits for UA
unsigned char waitAnswer();

//Opens Connection
int llopen(LinkLayer connectionParameters);

//Writes Data
int llwrite(const unsigned char *buf, int bufSize);

//Reads Data
int llread(unsigned char *packet);

//Closes Connection
int llclose(int showStatistics);
```

Main Use Cases

The Transmitter and the Receiver are the two different ways the program can be run as. For the Transmitter, the functions run are in this order:

1. llopen(): Initialises the connection sending the connection packet and waiting for the response from the receiver;
2. llwrite(): Creates and sends the data packets of the file we want to transfer from one computer to another;
3. waitAnswer(): Waits for the answer from the receiver after sending the packet, to see if the packet was well received by the target;
4. alarmHandler(): If the transmitter doesn't receive the answer during the seconds for the alarm, it triggers this function and resends the packet.
5. llclose(): Closes the connection between the two computers: sends a control packet, waits for the response, responds and closes the port.

The Receiver runs simultaneously and uses the functions in this order:

1. llopen(): Waits for the transmitter to send the SET packet that determines if the receiver should connect and start receiving data and sends the Acknowledgement if the frames were correct.
2. llread(): Receives a packet and analyses to check if it was correctly transferred. If not, a rejection message is sent and the sender retransmits the packet to be analysed again. If correct, the receiver sends a confirmation and continues receiving the rest of the packets, building the file.
3. llclose(): Closes the connection after receiving the control packet and sending an acknowledgement.

Logical Link Protocol

The `link_layer.c` file implements the core functions of the Link Layer Protocol (LLP) that manage reliable data transmission over a serial connection. This layer follows the Stop-and-Wait strategy, ensuring data integrity through mechanisms for connection establishment, data transfer, and disconnection.

The `llopen` function initiates the connection based on the role (transmitter or receiver). If in transmitter mode, it sends a "SET" frame and waits for a "UA" (Unnumbered Acknowledgement) frame from the receiver. This exchange acts as a handshake, aligning both devices before data transfer. For the receiver, `llopen` waits for a "SET" frame from the transmitter and responds with "UA," confirming that it is ready to receive data.

Once the connection is established, `llwrite` manages data frame transmission. It constructs a frame with sequence numbers and error-checking fields, then sends it to the receiver and waits for an acknowledgment (ACK). If no ACK is received within a timeout, `llwrite` retransmits the frame, adhering to the Stop-and-Wait protocol's requirement for error recovery. The function keeps retrying until an acknowledgment is received or it reaches a maximum retry limit, after which it returns an error.

On the receiving side, `llread` waits for and processes incoming frames. After checking each frame for errors, it extracts the data if the frame is valid and in sequence, sending an ACK back to the transmitter. If a frame is corrupted or out of sequence, `llread` discards it without sending an ACK, prompting the transmitter to retransmit the frame. This method ensures only valid, non-duplicate data is passed up to the application layer.

Finally, `llclose` gracefully terminates the communication, with the transmitter and receiver exchanging "DISC" (Disconnect) frames, followed by a "UA" acknowledgment to confirm disconnection. Afterward, `closeSerialPort` is called to release the serial port and restore its original settings.

Application Protocol

The `application_layer.c` file implements the Application Layer protocol, handling file transfers over a serial connection and managing different roles within the transmission (transmitter or receiver). The protocol begins by identifying the device's role—either as `LlTx` (transmitter) or `LlRx` (receiver)—and establishing the necessary connection parameters, including baud rate, number of retransmissions, and timeout duration. These parameters are stored within a `LinkLayer` structure and passed to the `llopen()` function to initiate the link.

For the **transmitter role** (`LlTx`), the protocol opens the specified file and prepares it for transfer by creating a "Start" Control Packet. This packet includes the file size and filename, which are added as metadata. After sending this initial packet with `llwrite()`, the protocol begins transferring data in segments by dividing the file into data packets. Each data packet consists of a control field, sequence number, size of data, and the data payload itself. The protocol manages each data packet in a loop, adjusting the remaining file size and advancing the sequence number as it progresses through the file. This loop enables the systematic transfer of the file content, ensuring all data packets are sent sequentially.

Once all data packets are sent, the transmitter sends an "End" Control Packet to signal the completion of the file transfer. This packet, similar to the "Start" packet, includes metadata, and its receipt signifies the end of the data transmission phase.

For the **receiver role** (`LlRx`), the protocol first waits to receive the "Start" Control Packet, which provides the file metadata required to initialise the file creation process on the receiving end. The packet is parsed to extract the file size and filename, and the receiver creates a new file using these details. Subsequently, the receiver enters a loop where it reads incoming data packets using `llread()`, parsing each packet to extract the data

payload and write it to the file. This process continues until an “End” Control Packet is received, signalling the successful completion of the file transfer.

The protocol concludes with a call to `llclose()` to properly close the connection, ensuring that any allocated resources are freed.

Validation

The program was subject to multiple tests, both by us and our professors. We tested running the following tests:

- File transfer with different names, types and sizes
- File transfer interrupted by the cable and resumed
- Initialising the Receiver without the Transmitter
- Initialising the Transmitter without the Receiver
- Stopping the Transmitter
- Stopping the Receiver
- Interruption from the physical connection and added noise

These tests allowed us to verify the resilience of the code and our program, all being successful and delivering the file without any errors.

Data Link Protocol Efficiency

Baud Rate Change

These are the results of some tests we conducted to study how a baud rate change would affect the efficiency of our program. A file with 10968 bytes was used during the testing process.

Baud Rate	Time (s)	Efficiency
1200	16.16	56.6%
4800	15.48	14.8%
9600	13.58	8.4%
19200	15.94	3.6%

As observed in the table above, the efficiency of our program is at its best when executed at lower baud rates, since the transfer times are way higher than expected for higher baud rates.

File Size Change

These are the results of some tests we conducted to study how a file size change would affect the efficiency of our program. The baud rate used in these tests was 1200.

File Size	Time (s)	Efficiency
6545	9.64	56.6%
10968	16.16	56.6%
12615	18.64	56.4%
18255	25.00	60.9%
28318	39.75	59.4%

The results suggest that our code is designed in a way that can handle files of varying sizes without compromising its efficiency.

Conclusions

In conclusion, this project provided practical insights into developing a Data Link Protocol using RS-232 and the Stop-and-Wait strategy. By implementing modular, layered code in the Link and Application Layers, we achieved reliable data transmission with error control, managing serial communication complexities such as timeouts and retransmissions. The Link Layer's role in ensuring data integrity highlighted the importance of error-handling mechanisms, while the Application Layer facilitated structured data handling, managing control and data packets for smooth, organised transmission.

Testing under various conditions, including interruptions, mismatches in sender-receiver roles, and added noise, validated the protocol's resilience and adaptability. Analysing the effects of baud rates and file sizes on efficiency demonstrated real-world trade-offs in transmission speed and accuracy. Overall, this project strengthened our understanding of protocol design, layering, and modularity, reinforcing the benefits of structured, isolated components in building robust communication protocols.

Appendix I - Source Code

serial_port.h

```
// Serial port header.
// NOTE: This file must not be changed.

#ifndef _SERIAL_PORT_H_
#define _SERIAL_PORT_H_

// Open and configure the serial port.
// Returns -1 on error.
int openSerialPort(const char *serialPort, int baudRate);

// Restore original port settings and close the serial port.
// Returns -1 on error.
int closeSerialPort();

// Wait up to 0.1 second (VTIME) for a byte received from the serial port (must
// check whether a byte was actually received from the return value).
// Returns -1 on error, 0 if no byte was received, 1 if a byte was received.
int readByteSerialPort(unsigned char *byte);

// Write up to numBytes to the serial port (must check how many were actually
// written in the return value).
// Returns -1 on error, otherwise the number of bytes written.
int writeBytesSerialPort(const unsigned char *bytes, int numBytes);

#endif // _SERIAL_PORT_H_
```

serial_port.c

```
// Serial port interface implementation
// DO NOT CHANGE THIS FILE

#include "serial_port.h"

#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <termios.h>
#include <unistd.h>

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

int fd = -1;           // File descriptor for open serial port
struct termios oldtio; // Serial port settings to restore on closing

// Open and configure the serial port.
// Returns -1 on error.
```

```
int openSerialPort(const char *serialPort, int baudRate)
{
    // Open with O_NONBLOCK to avoid hanging when CLOCAL
    // is not yet set on the serial port (changed later)
    int oflags = O_RDWR | O_NOCTTY | O_NONBLOCK;
    fd = open(serialPort, oflags);
    if (fd < 0)
    {
        perror(serialPort);
        return -1;
    }

    // Save current port settings
    if (tcgetattr(fd, &oldtio) == -1)
    {
        perror("tcgetattr");
        return -1;
    }

    // Convert baud rate to appropriate flag
    tcflag_t br;
    switch (baudRate)
    {
    case 1200:
        br = B1200;
        break;
    case 1800:
        br = B1800;
        break;
    case 2400:
        br = B2400;
        break;
    case 4800:
        br = B4800;
        break;
    case 9600:
        br = B9600;
        break;
    case 19200:
        br = B19200;
        break;
    case 38400:
        br = B38400;
        break;
    case 57600:
        br = B57600;
        break;
    case 115200:
        br = B115200;
        break;
    default:
        fprintf(stderr, "Unsupported baud rate (must be one of 1200, 1800, 2400,
4800, 9600, 19200, 38400, 57600, 115200)\n");
        return -1;
    }

    // New port settings
    struct termios newtio;
    memset(&newtio, 0, sizeof(newtio));

    newtio.c_cflag = br | CS8 | CLOCAL | CREAD;
    newtio.c_iflag = IGNPAR;
    newtio.c_oflag = 0;

    // Set input mode (non-canonical, no echo,...)
    newtio.c_lflag = 0;
    newtio.c_cc[VTIME] = 1; // Block reading
    newtio.c_cc[VMIN] = 0;  // Byte by byte
```

```
    tcflush(fd, TCIOFLUSH);

    // Set new port settings
    if (tcsetattr(fd, TCSANOW, &newtio) == -1)
    {
        perror("tcsetattr");
        close(fd);
        return -1;
    }

    // Clear O_NONBLOCK flag to ensure blocking reads
    oflags ^= O_NONBLOCK;
    if (fcntl(fd, F_SETFL, oflags) == -1)
    {
        perror("fcntl");
        close(fd);
        return -1;
    }

    // Done
    return fd;
}

// Restore original port settings and close the serial port.
// Returns -1 on error.
int closeSerialPort()
{
    // Restore the old port settings
    if (tcsetattr(fd, TCSANOW, &oldtio) == -1)
    {
        perror("tcsetattr");
        return -1;
    }

    return close(fd);
}

// Wait up to 0.1 second (VTIME) for a byte received from the serial port (must
// check whether a byte was actually received from the return value).
// Returns -1 on error, 0 if no byte was received, 1 if a byte was received.
int readByteSerialPort(unsigned char *byte)
{
    return read(fd, byte, 1);
}

// Write up to numBytes to the serial port (must check how many were actually
// written in the return value).
// Returns -1 on error, otherwise the number of bytes written.
int writeBytesSerialPort(const unsigned char *bytes, int numBytes)
{
    return write(fd, bytes, numBytes);
}
```

link_layer.h

```
// Link layer header.
// NOTE: This file must not be changed.

#ifndef _LINK_LAYER_H_
#define _LINK_LAYER_H_

typedef enum
{
   LlTx,
   LlRx,
} LinkLayerRole;

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



// SIZE of maximum acceptable payload.
// Maximum number of bytes that application layer should send to link layer
#define MAX_PAYLOAD_SIZE 1000

// MISC
#define FALSE 0
#define TRUE 1

// Open a connection using the "port" parameters defined in struct linkLayer.
// Return "1" on success or "-1" on error.
int llopen(LinkLayer connectionParameters);

// Send data in buf with size bufSize.
// Return number of chars written, or "-1" on error.
int llwrite(const unsigned char *buf, int bufSize);

// Receive data in packet.
// Return number of chars read, or "-1" on error.
int llread(unsigned char *packet);

// Close previously opened connection.
// if showStatistics == TRUE, link layer should print statistics in the console on
close.
// Return "1" on success or "-1" on error.
int llclose(int showStatistics);

#endif // _LINK_LAYER_H_
```

link_layer.c

```
// Link layer protocol implementation

#include "link_layer.h"
#include "serial_port.h"

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

// MISC
#define _POSIX_SOURCE 1 // POSIX compliant source
#define FLAG 0x7E
#define ESC 0x7D
#define A_Tx 0x03
#define A_Rx 0x01
#define C_SET 0x03
#define C_UA 0x07
#define C_DISC 0x0B

#define C_RR(Nr) (0xAA | (Nr & 1))
#define C_REJ(Nr) (0x54 | (Nr & 1))
#define C_N(Ns) (Ns << 7)
int Ns = 0;
int Nr = 1;

//GLOBAL VARIABLES
int alarmCount = 0;
int alarmEnabled = FALSE;
int timeout = 0;
int retransmitions = 0;
int role;


typedef enum
{
   START,
   STOP,
   FLAG_SM,
   A_SM,
   C_SM,
   BCC_SM,
   READ_SM,
   ESC_SM
} LinkLayerSM;

////////////////////////////////////////////////
// alarmHandler
////////////////////////////////////////////////
void alarmHandler(int signal)
{
   alarmEnabled = FALSE;  //this means that the alarm has been triggered
   alarmCount++;

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

   if (alarmCount > retransmitions)
```

```
    {
        return;
    }
}

////////////////////////////////////////////////
// waitAnswer
////////////////////////////////////////////////

unsigned char waitAnswer()
{

    unsigned char byte, ret = 0;
    LinkLayerSM state = START;

    while (state != STOP && alarmEnabled == TRUE)
    {
        readByteSerialPort(&byte);
        switch (state)
        {
            case START:
                if (byte == FLAG) state = FLAG_SM;
                break;
            case FLAG_SM:
                if (byte == A_Rx) state = A_SM;
                else if (byte != FLAG) state = START;
                break;
            case A_SM:
                if (byte == C_RR(0) || byte == C_RR(1) || byte == C_REJ(0) || byte
== C_REJ(1) || byte == C_DISC)
                {
                    state = C_SM;
                    ret = byte;
                }
                else if (byte == FLAG) state = FLAG_SM;
                else state = START;
                break;
            case C_SM:
                if (byte == (A_Rx ^ ret)) state = BCC_SM;
                else if (byte == FLAG) state = FLAG_SM;
                else state = START;
                break;
            case BCC_SM:
                if (byte == FLAG)
                {
                    state = STOP;
                }
                else state = START;
                break;
            default:
                break;
        }
    }
    return ret;
}


////////////////////////////////////////////////
// LLOPEN
////////////////////////////////////////////////
int llopen(LinkLayer connectionParameters)
{
    if (openSerialPort(connectionParameters.serialPort,
connectionParameters.baudRate)<0) return -1;
    LinkLayerSM state = START;
    int fd;
    unsigned char byte;
    timeout = connectionParameters.timeout;
```

```
retransmitions = connectionParameters.nRetransmissions;
role = connectionParameters.role == LlTx ? 0 : 1;

switch (connectionParameters.role)
{
    case LlTx:
    {
        (void) signal(SIGALRM, alarmHandler);
        int tries = 0;
        while (tries < retransmitions && state != STOP)
        {
            unsigned char message[5] = {FLAG, A_Tx, C_SET, A_Tx ^ C_SET, FLAG};
            if (writeBytesSerialPort(message, 5) == -1)
            {
                printf("Error writing SET\n");
                return -1;
            }
            printf("Wrote SET\n");
            alarm(timeout);
            alarmEnabled = TRUE;
            while(alarmEnabled == TRUE && state != STOP)
            {
                byte = 0;
                if (readByteSerialPort(&byte)) {
                    switch (state)
                    {
                    case START:
                        if(byte == FLAG)
                        {
                            state = FLAG_SM;
                        }
                        break;
                    case FLAG_SM:
                        if(byte == A_Rx)
                        {
                            state = A_SM;
                        } else if (byte != FLAG)
                        {
                            state = START;
                        }
                        break;
                    case A_SM:
                        if(byte == C_UA)
                        {
                            state = C_SM;
                        } else if (byte == FLAG)
                        {
                            state = FLAG_SM;
                        } else {
                            state = START;
                        }
                        break;
                    case C_SM:
                        if(byte == (A_Rx ^ C_UA))
                        {
                            state = BCC_SM;
                        } else if (byte == FLAG)
                        {
                            state = FLAG_SM;
                        } else
                        {
                            state = START;
                        }
                        break;
                    case BCC_SM:
                        if(byte == FLAG)
                        {
                            state = STOP;
```

```
                        alarm(0);
                        alarmEnabled = FALSE;

                    } else
                    {
                        state = START;
                    }
                    break;
                default:
                    break;
                }
            }
        }
        tries++;
    }
    if (state != STOP)
    {
        return -1;
    }
    printf("Received UA\n");
    fd = 0;
    break;
}
case LlRx:
{
    while (state != STOP)
    {
        byte = 0;
        if (readByteSerialPort(&byte))
        {
            switch (state)
            {
            case START:

                if(byte == FLAG)
                {
                    state = FLAG_SM;
                }
                break;
            case FLAG_SM:
                if(byte == A_Tx)
                {
                    state = A_SM;
                } else if (byte != FLAG)
                {
                    state = START;
                }
                break;
            case A_SM:
                if(byte == C_SET)
                {
                    state = C_SM;
                } else if (byte == FLAG)
                {
                    state = FLAG_SM;
                } else
                {
                    state = START;
                }
                break;
            case C_SM:
                if(byte == (A_Tx ^ C_SET))
                {
                    state = BCC_SM;
                } else if (byte == FLAG)
                {
                    state = FLAG_SM;
                } else
```

```
                        {
                            state = START;
                        }
                        break;
                    case BCC_SM:
                        if(byte == FLAG)
                        {
                            state = STOP;
                        } else
                        {
                            state = START;
                        }
                        break;
                    default:
                        break;
                    }
                }
            }
            printf("Received SET\n");

            unsigned char message[5] = {FLAG, A_Rx, C_UA, A_Rx ^ C_UA, FLAG};
            if (writeBytesSerialPort(message, 5) == -1)
            {
                printf("Error writing UA\n");
                return -1;
            }
            printf("Wrote UA\n");
            fd = 1;
            break;
        }
        default:
            printf("error in role");
            return -1;
    }
    return fd;
}

////////////////////////////////////////////////
// LLWRITE
////////////////////////////////////////////////
int llwrite(const unsigned char *buf, int bufSize)
{
    int frames = 6+bufSize;
    unsigned char *frame = (unsigned char *) malloc(frames);
    frame[0] = FLAG;
    frame[1] = A_Tx;
    frame[2] = C_N(Ns);
    frame[3] = frame[1] ^frame[2];

    int j = 4;
    for (unsigned int i = 0 ; i < bufSize ; i++) {
        if(buf[i] == FLAG || buf[i] == ESC) {
            frame = realloc(frame,++frames);
            frame[j++] = ESC;
            frame[j++] = buf[i] ^ 0x20;
        }
        else frame[j++] = buf[i];
    }
    unsigned char BCC = buf[0];
    for (unsigned int i = 1 ; i < bufSize ; i++) BCC ^= buf[i];
    frame[j++] = BCC;
    frame[j++] = FLAG;
    int tries = 0;
    int rej = 0;
    int ack = 0;
    while (tries < retransmitions)
    {
        alarmEnabled = FALSE;
```

```
        alarm(timeout);
        alarmEnabled = TRUE;
        while (alarmEnabled == TRUE && !rej && !ack)
        {
            if (writeBytesSerialPort(frame, frames) == -1)
            {
                printf("Error writing frame\n");
                return -1;
            }
            unsigned char answer = waitAnswer();
            if(!answer)
            {
                continue;
            }
            if (answer == C_RR(0) || answer == C_RR(1))
            {
                ack = 1;
                Ns = !Ns;
            }
            else if (answer == C_REJ(0) || answer == C_REJ(1))
            {
                rej = 1;
            }
            else if (answer == C_DISC)
            {
                return -1;
            }
            else
                continue;
        }
        if (ack)
        {
            printf("Frame sent successfully\n");
            break;
        }
        else if (rej)
        {
            printf("Error: need retransmition\n");
            rej = 0;
        }
        else {
            tries++;
            printf("Tries: %d\n", tries);
        }

    }
    free(frame);
    if(ack) return 0;
    return -1;
}



////////////////////////////////////////////////
// LLREAD
////////////////////////////////////////////////
int llread(unsigned char *packet)
{
    LinkLayerSM state = START;
    unsigned char byte, ret = 0;
    int i = 0;
    int n = 0;
    while (state != STOP)
    {
        if (readByteSerialPort(&byte))
        {
            switch (state)
            {
```

```
                case START:
                    if (byte == FLAG) state = FLAG_SM;
                    break;
                case FLAG_SM:
                    if (byte == A_Tx) state = A_SM;
                    else if (byte != FLAG) state = START;
                    break;
                case A_SM:
                    if (byte == C_N(0) || byte == C_N(1))
                    {
                        state = C_SM;
                        ret = byte;
                    }
                    else if (byte == FLAG) state = FLAG_SM;
                    else if (byte == C_DISC)
                    {
                        unsigned char message[5] = {FLAG, A_Rx, C_DISC, A_Rx ^
C_DISC, FLAG};
                        writeBytesSerialPort(message, 5);
                        closeSerialPort();
                        return 0;
                    }
                    else state = START;
                    break;
                case C_SM:
                    if (byte == (A_Tx ^ ret)) state = READ_SM;
                    else if (byte == FLAG) state = FLAG_SM;
                    else state = START;
                    break;
                case READ_SM:
                    if (byte == ESC) state = ESC_SM; //vai desfazer o byte stuffing
                    else if (byte == FLAG) //termina o pacote
                    {
                        unsigned char bcc2 = packet[i-1];
                        i--;
                        packet[i] = '\0';
                        unsigned char acc = packet[0];

                        for (unsigned int j = 1; j < i; j++)
                            acc ^= packet[j];

                        if (bcc2 == acc)
                        {
                            state = STOP;
                            unsigned char message[5] = {FLAG, A_Rx, C_RR(Nr), A_Rx ^
C_RR(Nr), FLAG};
                            writeBytesSerialPort(message, 5);
                            printf("Read a packet\n");
                            Nr = !Nr;
                            return i;
                        }
                        else
                        {
                            printf("Error: need retransmition\n");
                            unsigned char message[5] = {FLAG, A_Rx, C_REJ(Nr), A_Rx
^ C_REJ(Nr), FLAG};
                            writeBytesSerialPort(message, 5);
                            return -1;
                        };

                    }
                    else
                    {
                        packet[i++] = byte; //escreve o byte no pacote
                    }
                    break;
                case ESC_SM:
                    state = READ_SM;
```

```
                    packet[i++] = byte ^ 0x20;
                    break;
                default:
                    break;

            }
            n++;
        }

    }

    return i;
}

////////////////////////////////////////////////
// LLCLOSE
////////////////////////////////////////////////
int llclose(int showStatistics)
{
    LinkLayerSM state = START;
    unsigned char byte;

    switch (role)
    {
        case 0:
        {
            (void) signal(SIGALRM, alarmHandler);
            int tries = 0;
            while (tries < retransmitions && state != STOP)
            {
                unsigned char message[5] = {FLAG, A_Tx, C_DISC, A_Tx ^ C_DISC,
FLAG};
                if (writeBytesSerialPort(message, 5) == -1)
                {
                    printf("Error writing SET\n");
                    return -1;
                }
                printf("Wrote DISC\n");
                alarm(timeout);
                alarmEnabled = TRUE;
                while(alarmEnabled == TRUE && state != STOP)
                {
                    byte = 0;
                    if (readByteSerialPort(&byte)) {
                        switch (state)
                        {
                        case START:
                            if(byte == FLAG)
                            {
                                state = FLAG_SM;
                            }
                            break;
                        case FLAG_SM:
                            if(byte == A_Rx)
                            {
                                state = A_SM;
                            } else if (byte != FLAG)
                            {
                                state = START;
                            }
                            break;
                        case A_SM:
                            if(byte == C_UA)
                            {
                                state = C_SM;
                            } else if (byte == FLAG)
                            {
                                state = FLAG_SM;
```

```
                    } else {
                        state = START;
                    }
                    break;
                case C_SM:
                    if(byte == (A_Rx ^ C_UA))
                    {
                        state = BCC_SM;
                    } else if (byte == FLAG)
                    {
                        state = FLAG_SM;
                    } else
                    {
                        state = START;
                    }
                    break;
                case BCC_SM:
                    if(byte == FLAG)
                    {
                        state = STOP;
                        alarm(0);
                        alarmEnabled = FALSE;

                    } else
                    {
                        state = START;
                    }
                    break;
                default:
                    break;
                }
            }
        }
        tries++;
    }
    if (state != STOP)
    {
        return -1;
    }
    printf("Received DISC\n");
    unsigned char message[5] = {FLAG, A_Tx, C_UA, A_Tx ^ C_UA, FLAG};
    if (writeBytesSerialPort(message, 5) == -1)
    {
        printf("Error writing SET\n");
        return -1;
    }
    printf("Wrote UA\n");

    printf("%d frames were sent successfully!\n", showStatistics);

    break;
}
case 1:
{
    while (state != STOP)
    {
        byte = 0;
        if (readByteSerialPort(&byte))
        {
            switch (state)
            {
            case START:

                if(byte == FLAG)
                {
                    state = FLAG_SM;
                }
                break;
```

```
                    case FLAG_SM:
                        if(byte == A_Tx)
                        {
                            state = A_SM;
                        } else if (byte != FLAG)
                        {
                            state = START;
                        }
                        break;
                    case A_SM:
                        if(byte == C_DISC)
                        {
                            state = C_SM;
                        } else if (byte == FLAG)
                        {
                            state = FLAG_SM;
                        } else
                        {
                            state = START;
                        }
                        break;
                    case C_SM:
                        if(byte == (A_Tx ^ C_DISC))
                        {
                            state = BCC_SM;
                        } else if (byte == FLAG)
                        {
                            state = FLAG_SM;
                        } else
                        {
                            state = START;
                        }
                        break;
                    case BCC_SM:
                        if(byte == FLAG)
                        {
                            state = STOP;
                        } else
                        {
                            state = START;
                        }
                        break;
                    default:
                        break;
                    }
                }
            }

            printf("Received DISC\n");

            unsigned char message[5] = {FLAG, A_Rx, C_UA, A_Rx ^ C_UA, FLAG};
            if (writeBytesSerialPort(message, 5) == -1)
            {
                printf("Error writing UA\n");
                return -1;
            }
            printf("Wrote UA\n");
            clock_t end = clock();
            double time_spent = (double)(end - (clock_t) showStatistics) /
CLOCKS_PER_SEC;
            printf("The program ran for %f seconds!\n", time_spent * 42);
            break;
        }
        default:
            printf("error in role");
            return -1;
    }
```

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

application_layer.h

```
// Application layer protocol header.
// NOTE: This file must not be changed.

#ifndef _APPLICATION_LAYER_H_
#define _APPLICATION_LAYER_H_

// Application layer main function.
// Arguments:
//   serialPort: Serial port name (e.g., /dev/ttyS0).
//   role: Application role {"tx", "rx"}.
//   baudrate: Baudrate of the serial port.
//   nTries: Maximum number of frame retries.
//   timeout: Frame timeout.
//   filename: Name of the file to send / receive.
void applicationLayer(const char *serialPort, const char *role, int baudRate,
                      int nTries, int timeout, const char *filename);

#endif // _APPLICATION_LAYER_H_
```

application_layer.c

```
// Application layer protocol implementation

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


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


clock_t start, end;


void applicationLayer(const char *serialPort, const char *role, int baudRate,
                      int nTries, int timeout, const char *filename)
{
    printf("Starting application layer\n");
    start = clock();
    int info = 0;
    LinkLayerRole rolex;
    //Role
```

```
    switch (role[0])
    {
        case 't':
            rolex = LlTx;
            break;
        case 'r':
            rolex = LlRx;
            break;
        default:
            printf("Invalid Role\n");
            return;
    }

    //LinkLayer
    LinkLayer connectionParameters = {
        .role = rolex,
        .baudRate = baudRate,
        .nRetransmissions = nTries,
        .timeout = timeout
    };
    strcpy(connectionParameters.serialPort,serialPort);

    printf("Set connection settings\n");

    //llopen
    int error1 = llopen(connectionParameters);
    if (error1 == -1)
    {
        printf("Error in llopen\n");
        return;
    }
    printf("Established connection\n");
    switch (rolex)
    {
    case LlTx:
        FILE* file = fopen(filename, "rb");
        if (file == NULL)
        {
            printf("Error opening file\n");
            return;
        }
        printf("File opened\n");

        //prepare Control Packet 1
        fseek(file, 0, SEEK_END);
        long int fileSize = ftell(file);
        fseek(file, 0, SEEK_SET);

        printf("File size: %ld\n", fileSize);

        //create Control Packet 1
        unsigned int cpSize;
        const int L1 = sizeof(fileSize);
        const int L2 = strlen(filename);
        cpSize = 1+2+L1+2+L2;
        unsigned char *controlPacketStart = (unsigned char*)malloc(cpSize);

        unsigned int pos = 0;
        controlPacketStart[pos++]=1; //start
        controlPacketStart[pos++]=0; //type (file size)
        controlPacketStart[pos++]=L1; //length (file size)

        for (int i = 0; i < L1; i++)
        {
            controlPacketStart[pos + i] = (fileSize >> (8 * (L1 - 1 - i))) & 0xFF;
// MSB first
        }
        pos += L1; // Move the position forward
```

```
        controlPacketStart[pos++]=1; //type (file name)
        controlPacketStart[pos++]=L2; //length (file name)
        memcpy(controlPacketStart+pos, filename, L2); //file name

        //write Control Packet 1
        if (llwrite(controlPacketStart, cpSize)<0)
        {
            printf("Error writing Control Packet 1\n");
            return;
        }

        info++;
        //prepare Data Packets
        unsigned char sequence = 0;
        long int bytesLeft = fileSize;

        unsigned char* content = (unsigned char*)malloc(sizeof(unsigned char) *
fileSize);
        fread(content, sizeof(unsigned char), fileSize, file);

        while (bytesLeft > 0)
        {
            //create Data Packet
            int dataSize = bytesLeft > (long int) MAX_PAYLOAD_SIZE ?
MAX_PAYLOAD_SIZE : bytesLeft;
            unsigned char* data = (unsigned char*) malloc(dataSize);
            memcpy(data, content, dataSize);

            int packetSize = 1 + 1 + 2 + dataSize;
            unsigned char* packet = (unsigned char*)malloc(packetSize);

            packet[0] = 2; //control field
            packet[1] = sequence; //sequence number
            //256 * L2 + L1
            packet[2] = (dataSize >> 8) & 0xFF; //L2
            packet[3] = dataSize & 0xFF;        //L1
            memcpy(packet+4, data, dataSize);


            if(llwrite(packet, packetSize) == -1)
            {
                printf("Exit: error in data packets\n");
                exit(-1);
            }
            bytesLeft -= (long int) MAX_PAYLOAD_SIZE;
            content += dataSize;
            sequence = (sequence + 1) % 255;
            info++;
        }


        //create Control Packet 3
        unsigned char *controlPacketEnd = (unsigned char*)malloc(cpSize);

        unsigned int sop = 0;
        controlPacketEnd[sop++]=3; //end
        controlPacketEnd[sop++]=0; //type (file size)
        controlPacketEnd[sop++]=L1; //length (file size)

        for (int i = 0; i < L1; i++)
        {
            controlPacketEnd[sop + i] = (fileSize >> (8 * (L1 - 1 - i))) & 0xFF; //
MSB first
        }
        sop += L1; // Move the position forward

        controlPacketEnd[sop++]=1; //type (file name)
```

```
        controlPacketEnd[sop++]=L2; //length (file name)
        memcpy(controlPacketEnd+sop, filename, L2); //file name

        //write Control Packet 3
        if (llwrite(controlPacketEnd, cpSize)<0)
        {
            printf("Error writing Control Packet 1\n");
            return;
        }

        info++;

        break;
    case LlRx:
        //create packet
        unsigned char *packet = (unsigned char *)malloc(MAX_PAYLOAD_SIZE);
        int packetSize = -1;
        while ((packetSize = llread(packet)) < 0) {}

        //parse Control Packet 1

        long rxFileSize = 0;
        unsigned char *name;

        // File Size
        unsigned char fileSizeNBytes = packet[2]; // Length of the file size (L1)

        // Dynamically allocate memory for the file size
        unsigned char* fileSizeAux = (unsigned char*)malloc(fileSizeNBytes);

        memcpy(fileSizeAux, packet + 3, fileSizeNBytes); // Copy the file size from
the packet

        for (unsigned int i = 0; i < fileSizeNBytes; i++) {
            rxFileSize |= (fileSizeAux[fileSizeNBytes - i - 1] << (8 * i)); //
Reconstruct the file size
        }


        // File Name
        unsigned char fileNameNBytes = packet[3 + fileSizeNBytes + 1]; // L2
        name = (unsigned char *)malloc(fileNameNBytes);

        memcpy(name, packet + 3 + fileSizeNBytes + 2, fileNameNBytes); // Copy file
name
        name[fileNameNBytes] = '\0'; // Null-terminate the string if needed


        FILE* newFile = fopen((char *) filename, "wb+");
        while (1) {
            while ((packetSize = llread(packet)) < 0);
            if(packetSize == 0) break;
            else if(packet[0] != 3){
                unsigned char *buffer = (unsigned char*)malloc(packetSize);

                //parse Data Packet
                memcpy(buffer,packet+4,packetSize-4);

                fwrite(buffer, sizeof(unsigned char), packetSize-4, newFile);
                free(buffer);
            }
            else break;
        }
        fclose(newFile);
        printf("%s created\n", filename);
        info = (int) start;
        break;
    default:
```

```
        printf("Invalid role\n");
        break;
    }

    if (llclose(info) < 0)
    {
        printf("Error on llclose");
        return;
    }
    printf("Exiting...\nGoodbye.\n");
}
```