Faculdade de Engenharia da Universidade do Porto

Data Link Protocol

'Redes de Computadores'

Class 12 Group 5

Gonçalo Sousa (up202207320) Participation: 50%

Angy Pita (up202007253) Participation: 50%

Summary

This project was developed in the '*Redes de Computadores*' UC context and the main goal was to ensure the implementation of the data communication protocol to transmit and receive data through Serial Port.

In this specific case we were requested to transfer an image '*.gif*' from a computer to another (or using the virtual cable interface provided locally).

Introduction

The report follows a strict structure divided into eight topics that we'll go deeper further. We are also going to append pieces of our code after the conclusion to brighten the logic we used.

Architecture

All the code we implemented is between the two layers of our project. Those are the *'LinkLayer'* and the *'ApplicationLayer'*, with the header files (*'link_layer.h'* and *'application_layer.h'*) inside the *'include'* folder and source files(*'link_layer.c'* and *'application_layer.c'*) inside the *'src'* folder.

LinkLayer - has most of the logical part, including the whole protocol we learnt during the classes adapted to this case scenario. Here we make, send and validate the data frames as well as handle any errors (if occurred) during the transmission. This layer's also responsible for the connection, establishing it or not as requested.

ApplicationLayer - makes use of the '*LinkLayer'* functionalities to handle the packets (receiving and sending). The user is also able to define the size of the frames, maximum number of retransmission and the transfer speed of data packets in this module.

Code Structure

ApplicationLayer

In the implementation of the *ApplicationLayer*, auxiliary functions were defined to manage file transmission and reception without requiring additional data structures:

`void` **`applicationLayer`**`(const char *serialPort, const char *mode, int baudRate, int maxRetries, int waitTime, const char *filename)-` Primary function for the initialization and control of the application layer, sets parameters, and calls transmission or reception functions based on the configured role (mode).

`static int` **`startTransmission`**`(const char *filename)-` Initiates the transmission of a specified file, manages control and data packets, and sends each data segment.

`static int` **`startReception`**`(const char *filename)-` Initiates file reception, writes received data to the target file, and verifies the final control packet to confirm the end of transmission.

`static FILE*` **`openFile`**`(const char *filename, const char *mode)-` Opens a file in the specified mode and returns the file pointer.

`static long` **`calculateFileSize`**`(FILE *file)-` Calculates the file size by moving the pointer to the end and then returning it to the beginning.

`static unsigned char*` **`createControlPacket`**`(unsigned char type, const char *filename, long fileSize)-` Creates an initial or final control packet containing the type, file size, and file name.

`static int` **`sendControlPacket`**`(unsigned char *packet, int packetSize)-` Sends a control packet and checks for successful transmission.

`static unsigned char*` **`createDataPacket`**`(unsigned char sequence, const unsigned char *data, int dataSize)-` Creates a data packet with the current sequence and the file data content.

`static unsigned char` **`getNextSequence`**`(unsigned char sequence)-` Updates the packet sequence, keeping it cyclic between 0 and 255.

LinkLayer

The *LinkLayer* implements a state machine that allows communication between two devices. Is in this file that we also take the statistics into account. For this we implemented five additional data structures:

```
typedef enum {
     FLAG = 0x7E,          //Usado para indicar o início e fim de
uma trama
    ESCAPE = 0x7D       //Para aplicar byte stuffing
} ControlCharacters;

// Endereços utilizados no protocolo
typedef enum {
    Address_Transmitter = 0x03,
    Address_Receiver = 0x01
} Address;

// Comandos de controlo para estabelecer e encerrar a comunicação
typedef enum {
    Command_SET = 0x03,
    Command_UA = 0x07,
    Command_DISC = 0x0B,     //Encerra a comunicação
    Command_DATA = 0x01,     //Indica envio de dados
    Command_RR = 0x05,      //Reconhece recebimento correto de uma
   trama
    Command_REJ = 0x01      //Rejeita uma trama incorreta
} ControlCommands;

typedef struct {
    int tramasEnviadas;
    int tramasRecebidas;
    int tramasRejeitadas;
    int tramasAceitas;
    int totalBytesTransmitidos;
    double tiempoTransmision;
    double tiempoRecepcion;
    double tiempoDesconexion;
    double tiempoTransferencia;
} EstatisticasConexao;
```

```
typedef enum {
    START,
    FLAG_RCV,
    A_RCV,
    C_RCV,
    BCC1_OK,
    DATA,
    STOP_R
} LinkLayerState;
```

The main functions developed for the *LinkLayer* are:

`int llopen(LinkLayer *connectionParameters)-` Establishes a communication connection based on the provided parameters, such as serial port and device role.

`int llwrite(const unsigned char *buf, int bufSize)-` Sends data frames using byte stuffing and includes integrity checks (BCC).

`int llread(unsigned char *packet)-` Reads received frames, applies byte destuffing, and checks the BCC values to ensure data integrity.

`int llclose(int showStatistics)-` Closes the connection and serial port, showing performance statistics like total connection time and protocol efficiency.

`void alarmHandler(int signal)-` Configures and manages the timeout alarm during transmissions.

`int applyByteStuffing(const unsigned char *input, int length, unsigned char *output)-` Scans each byte in the input array, replacing FLAG and ESCAPE bytes with specific escape sequences and outputs the modified array, returning the final length.

`int applyByteDestuffing(const unsigned char *input, int length, unsigned char *output)-` Processes each byte in the input array, reversing any escape sequences created by byte stuffing and outputs the restored data, returning the length of the destuffed array.

`int enviarTramaSupervisao(int fd, unsigned char A, unsigned char C)-` Sends a supervision frame with the specified address (A) and command (C) values, for control and acknowledgment operations.

`unsigned char calculateBCC2(const unsigned char *buf, int bufSize)-` Computes a simple error-checking code (BCC2) by performing a XOR operation.

Statistics Structures

`void` **`actualizarEstadisticasEnvio`**`(int aceito)`- Updates the transmission statistics.

`void` **`actualizarEstadisticasRecepcao`**`()`- Updates the reception statistics.

`void` **`mostrarEstatisticas`**`()`- Displays all the statistics.

Main Use Cases

Connection Establishment

`applicationLayer(serialPort, mode, baudRate, maxRetries, waitTime, filename):` Opens the application connection using the specified serial port and baud rate.

`llopen(config):` Initializes the communication link with the provided parameters.

Data Transmission

`startTransmission(filename):` Sends the file data.

`openFile(filename, "rb"):` Opens the file for reading.

`calculateFileSize(file):` Calculates the file size to include in the control packet.

`createControlPacket(type, filename, fileSize):` Creates the control packet.

`sendControlPacket(packet, packetSize):` Sends the control packet to the receiver.

`createDataPacket(sequence, buffer, bytesRead):` Creates data packets with the correct sequence number.

`llwrite(dataPacket, bytesRead + 4):` Sends the data packet to receiver.

`getNextSequence(sequence):` Updates the sequence number for the next data packet.

`sendControlPacket(controlPacket, packetSize):` Sends the final control packet indicating transmission completion.

`fclose(file):` Closes the file after transmission is complete.

Data Reception

`startReception(filename);` Receives the file data.

`openFile(filename, "wb");` Opens the file for writing.

`llread(buffer);` Reads the incoming data packets.

`fwrite(buffer + 4, sizeof(unsigned char), packetSize - 4, file);` Writes the received data to the file.

`fclose(file);` Closes the file after reception is complete.

Connection Closing

`llclose();` Closes the connection and releases the serial port.

Logical Link Protocol

In this project, this handles serial port communication between the transmitter and receiver using a *Stop-and-Wait* protocol. To establish the connection, the '*llopen'* function configures the serial port, then sends a *SET* frame from the transmitter and waits for a *UA* frame from the receiver. When the receiver responds with *UA*, the connection is confirmed, allowing data transmission to begin.

During data transmission, '*llwrite'* applies byte stuffing to prevent conflicts with frame flags, transforms the data into a frame, and sends it to the receiver. If the frame is not acknowledged, the function retries until either the frame is accepted or the retry limit is reached. Each transmission attempt has a timeout to handle delays.

Reception of data is managed by *'llread'*, which destuffs the data field, verifies integrity via *BCC1* and *BCC2* checks, and ensures that errors during transmission are caught and handled.

Connection termination occurs with '*llclose'*, which is called once all data is sent or the retry limit is exceeded. The transmitter sends a *DISC* frame, waits for a matching *DISC* from the receiver, and upon receiving *DISC* again, responds with a *UA* frame to finalize the connection and close the serial port.

Application Protocol

In the application protocol, file transfer between transmitter and receiver begins by configuring the communication parameters, including baud rate, maximum retransmissions number, and timeout. Depending on the selected mode, either '*startTransmission'* or '*startReception'* is invoked, and a serial link is established using '*llopen'*.

When transmitting a file, an initial control packet is sent containing essential file metadata - name and size - allowing the receiver to allocate the necessary resources. The file data is then read in chunks, formatted into data packets, and sent sequentially. Each data packet includes a sequence number to ensure order and detect any packet loss. After sending each packet, the transmitter waits for acknowledgement from the receiver; if it's not acknowledged, the packet is retransmitted up to the maximum retry limit.

On the receiving side, data packets are processed to extract the original file content. If a final control packet is received, the file transfer is complete. The connection is then closed using '*llclose'*, finalizing the file transfer process, or when the maximum retransmission number is exceeded.

Validation

In the making of this project we have made some tests to evaluate our code's effectiveness. Besides all the debugging we used to ensure the proper running flow of the program, we also tested the interruption (with *cable off* and *on,* virtually), even though it didn't work at first; transfer with different packet sizes and different baud rates.

We also ensured that our project was noise proof, so with the noise equal to 0.001 it worked, as we are going to show later in this report.

Data Link Protocol Efficiency

Baudrate variation

FILE SIZE : 10968 bytes (penguin.gif)		
Baudrate (bits/s)	**Time (s)**	**Efficiency (%)**
4800	25.4	73.2
9600	12.3	75.7
38400	3.2	71.6
57600	2.2	71.0
115200	1.3	59.0

As the baudrate increases, the efficiency decreases. Another factor we have to take into account is that the baudrate and the transfer time are inversely proportional.

Propagation Delay Variation

FILE SIZE : 10968 bytes (penguin.gif) // BAUDRATE: 9600		
Delay ($x10^{-6}$ s)	Time (s)	Efficiency (%)
1000	12.4	75.1
10000	13.3	69.9
50000	16.8	55.4
100000	21.4	43.4
1000000	104.2	8.9

As expected, as the delay increases, the transfer time also increases and consequently efficiency decreases.

BER Variation

FILE SIZE : 10968 bytes (penguin.gif) // BAUDRATE: 9600		
BER	Time (s)	Efficiency (%)
0,0001	40.2	23.1
0,00005	20.21	46.0
0,00001	17.1	54.5
0	12.3	75.7

As we add more noise to the data bits, the transfer time increases. The same doesn't happen, as expected, with the efficiency, that gets significantly worse with the noise.

Conclusion

In the end, this project turned out to be really useful for the better understanding of crucial and base concepts of data transfer, such as byte stuffing and destuffing and especially data framing.

Our group was able to transfer data (in this case, the penguin.gif) using the protocols learnt during the classes, implementing a 'noise proof' and cable on/off responsive code.

Appendix I - Header Files

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_
```

link_layer.h

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

#ifndef _LINK_LAYER_H_
#define _LINK_LAYER_H_
#define MAX_FRAME_SIZE 2048
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_
```

Appendix II - Source Files

application_layer.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <termios.h>
#include <time.h>
#include "application_layer.h"
#include "link_layer.h"
#include "serial_port.h"
```

```
// Declaración de funciones auxiliares utilizadas en la capa de aplicación
static int startTransmission(const char *filename);
static int startReception(const char *filename);
static FILE* openFile(const char *filename, const char *mode);
static long calculateFileSize(FILE *file);
static unsigned char* createControlPacket(unsigned char type, const char *filename, long fileSize);
static int sendControlPacket(unsigned char *packet, int packetSize);
static unsigned char* createDataPacket(unsigned char sequence, const unsigned char *data, int dataSize);
static unsigned char getNextSequence(unsigned char sequence);

////////////////////////////////////////////////
// APPLICATION LAYER - Gestor principal da camada de aplicação
////////////////////////////////////////////////
// Parâmetros:
//   serialPort: porta serial para comunicação
//   mode: "tx" para transmissão, "rx" para recepção
//   baudRate: taxa de transmissão
//   maxRetries: número de tentativas de retransmissão
//   waitTime: tempo limite de espera para retransmissão
//   filename: nome do arquivo a ser enviado ou recebido
void applicationLayer(const char *serialPort, const char *mode, int baudRate, int maxRetries, int waitTime, const char *filename) {
    // Configuração dos parâmetros para a camada de enlace
    LinkLayer config = {
        .baudRate = baudRate,
        .timeout = waitTime,
        .nRetransmissions = maxRetries,
        .role = strcmp(mode, "tx") == 0 ? LITx : LIRx,
    };
    strcpy(config.serialPort, serialPort);

    // Abre a conexão serial usando llopen
    if (llopen(config) < 0) {
        perror("Erro ao abrir conexão\n");
        exit(-1);
    }

    // Marca o tempo de início para medir a duração da transmissão
    clock_t start = clock();
```

```
        // Escolha entre transmissão e recepção, conforme o papel da conexão
        if (config.role == LITx) {
            if (startTransmission(filename) < 0) {
                perror("Erro durante a transmissão\n");
                exit(-1);
            }
        } else if (config.role == LIRx) {
            if (startReception(filename) < 0) {
                perror("Erro durante a recepção\n");
                exit(-1);
            }
        }

        // Calcula e exibe o tempo de transmissão total
        clock_t end = clock();
            printf("Tempo de transmissão: %.2f segundos\n", (double)(end - start) /
CLOCKS_PER_SEC);

        // Fecha a conexão serial
        llclose(1);
    }

    // Inicia a transmissão de um arquivo
    static int startTransmission(const char *filename) {
        // Abre o arquivo em modo de leitura
        FILE *file = openFile(filename, "rb");
        if (!file) return -1;

        // Calcula o tamanho do arquivo para incluir no pacote de controle
        long fileSize = calculateFileSize(file);

        // Envia o pacote de controle inicial com informações do arquivo
        unsigned char *controlPacket = createControlPacket(0x02, filename, fileSize);
        if (sendControlPacket(controlPacket, strlen((char *)controlPacket) + 1) < 0) {
            free(controlPacket);
            return -1;
        }
        free(controlPacket);

        // Variáveis para gerenciar sequência e buffer de dados
        unsigned char sequence = 0;
        unsigned char buffer[256];
        int bytesRead;
```

```
    // Lê e envia os dados do arquivo em pacotes de tamanho fixo até o final do arquivo
    while ((bytesRead = fread(buffer, 1, sizeof(buffer), file)) > 0) {
        // Cria o pacote de dados com o número de sequência atual
        unsigned char *dataPacket = createDataPacket(sequence, buffer, bytesRead);
        // Envia o pacote e verifica erros
        if (llwrite(dataPacket, bytesRead + 4) < 0) {
            free(dataPacket);
            return -1;
        }
        sequence = getNextSequence(sequence);  // Atualiza a sequência
        free(dataPacket);
    }

    // Envia o pacote de controle final indicando o término da transmissão
    controlPacket = createControlPacket(0x03, filename, fileSize);
    if (sendControlPacket(controlPacket, strlen((char *)controlPacket) + 1) < 0) {
        free(controlPacket);
        return -1;
    }
    free(controlPacket);

    // Fecha o arquivo após a transmissão completa
    fclose(file);
    return 0;
}

// Inicia a recepção de um arquivo
static int startReception(const char *filename) {
    // Abre o arquivo em modo de escrita
    FILE *file = openFile(filename, "wb");
    if (!file) return -1;

    unsigned char buffer[512];
    int packetSize;

    // Recebe pacotes até o final do arquivo (pacote de controle final)
    while ((packetSize = llread(buffer)) > 0) {
        if (buffer[0] == 0x01) {  // Verifica se é um pacote de dados
            fwrite(buffer + 4, sizeof(unsigned char), packetSize - 4, file);
        } else if (buffer[0] == 0x03) {  // Pacote de controle final
            break;
        }
```

```
        }

        fclose(file);  // Fecha o arquivo após a recepção completa
        return packetSize < 0 ? -1 : 0;
    }

    // Abre um arquivo com o modo especificado
    static FILE* openFile(const char *filename, const char *mode) {
        FILE *file = fopen(filename, mode);
        if (!file) {
            perror("Erro ao abrir o arquivo\n");
        }
        return file;
    }

    // Calcula o tamanho do arquivo
    static long calculateFileSize(FILE *file) {
        // Move o ponteiro do arquivo para o final para obter o tamanho
        fseek(file, 0, SEEK_END);
        long size = ftell(file);
        // Retorna o ponteiro ao início
        fseek(file, 0, SEEK_SET);
        return size;
    }

    // Cria um pacote de controle para iniciar ou terminar a transmissão
    static unsigned char* createControlPacket(unsigned char type, const char *filename,
long fileSize) {
        int filenameSize = strlen(filename);
        unsigned char *packet = malloc(7 + filenameSize);

        // Define o tipo de controle e comprimento do tamanho do arquivo
        packet[0] = type;
        packet[1] = 0;
        packet[2] = sizeof(long);

        // Insere o tamanho do arquivo no pacote
        for (int i = sizeof(long) - 1; i >= 0; i--) {
            packet[3 + i] = (fileSize >> (8 * i)) & 0xFF;
        }

        // Insere o nome do arquivo no pacote
        packet[3 + sizeof(long)] = 1;
```

```
        packet[4 + sizeof(long)] = filenameSize;
        memcpy(packet + 5 + sizeof(long), filename, filenameSize);

        return packet;
    }

    // Cria um pacote de dados com número de sequência
    static unsigned char* createDataPacket(unsigned char sequence, const unsigned char
*data, int dataSize) {
        unsigned char *packet = malloc(dataSize + 4);

        // Estrutura do pacote de dados: flag, sequência e tamanho
        packet[0] = 0x01;
        packet[1] = sequence;
        packet[2] = (dataSize >> 8) & 0xFF;
        packet[3] = dataSize & 0xFF;
        memcpy(packet + 4, data, dataSize);  // Adiciona os dados

        return packet;
    }

    // Envia um pacote de controle e verifica se foi bem-sucedido
    static int sendControlPacket(unsigned char *packet, int packetSize) {
        int result = llwrite(packet, packetSize);
        if (result < 0) {
            perror("Erro ao enviar o pacote de controle\n");
        }
        return result;
    }

    // Atualiza o número de sequência
    static unsigned char getNextSequence(unsigned char sequence) {
        return (sequence + 1) % 256;  // Garante que o número de sequência é cíclico entre 0
e 255
    }
```

link_layer.c

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <signal.h>
    #include <unistd.h>
    #include <time.h>
    #include "serial_port.h"
```

```
#include "link_layer.h"

#define C_RR0 0xAA   // RR0: el receptor está listo para recibir la trama de información número 0
#define C_RR1 0xAB   // RR1: el receptor está listo para recibir la trama de información número 1
#define C_REJ0 0x54  // REJ0: el receptor rechaza la trama de información número 0 (se detectó un error)
#define C_REJ1 0x55  // REJ1: el receptor rechaza la trama de información número 1 (se detectó un error)
int tramaRx = 0;
// Enums para caracteres de controle e comandos de comunicação
typedef enum {
    FLAG = 0x7E,        //Usado para indicar o início e fim de uma trama
    ESCAPE = 0x7D       //Para aplicar byte stuffing
} ControlCharacters;

// Endereços utilizados no protocolo
typedef enum {
    Address_Transmitter = 0x03,
    Address_Receiver = 0x01
} Address;

// Comandos de controlo para estabelecer e encerrar a comunicação
typedef enum {
    Command_SET = 0x03,
    Command_UA = 0x07,
    Command_DISC = 0x0B,    //Encerra a comunicação
    Command_DATA = 0x01,    //Indica envio de dados
    Command_RR = 0x05,      //Reconhece recebimento correto de uma trama
    Command_REJ = 0x01      //Rejeita uma trama incorreta
} ControlCommands;

unsigned char frame[MAX_FRAME_SIZE];        // Array para armazenar uma trama temporária

extern int fd;
LinkLayerRole currentRole;  //Transmissor ou receptor
LinkLayer param;
// Estrutura para armazenar as estatísticas de conexão
typedef struct {
     int tramasEnviadas;
    int tramasRecebidas;
```

```
    int tramasRejeitadas;
    int tramasAceitas;
    int totalBytesTransmitidos;
    double tiempoTransmision;
    double tiempoRecepcion;
    double tiempoDesconexion;
    double tiempoTransferencia;
} EstatisticasConexao;

// Instância global para as estatísticas
EstatisticasConexao estatisticas = {0, 0, 0, 0, 0.0, 0.0, 0.0};

// Estados utilizados na máquina de estados para o protocolo de ligação
typedef enum {
    START,
    FLAG_RCV,
    A_RCV,
    C_RCV,
    BCC1_OK,
    DATA,
    STOP_R
} LinkLayerState;

int timeout = 3;
int retransmissions = 5;
int alarmEnabled = 0;
int alarmCount = 0;
clock_t desconexionStart;

clock_t conexionStart;

// Função de tratamento da interrupção de alarme
void alarmHandler(int signal) {
    alarmEnabled = 1;
    alarmCount++;
}

// Função para enviar uma trama de supervisão (controlo)
/*int enviarTramaSupervisao1(unsigned char address, unsigned char control) {
    unsigned char frame[5] = {0}; // Criação da trama de controlo
    frame[0] = FLAG;
    frame[1] = address;
    frame[2] = control;
```

```
        frame[3] = address ^ control;
        frame[4] = FLAG;
        int bytes = writeBytesSerialPort(frame, 5);
        if(bytes < 0){
            printf("Erro  ao enviar trama de supervisão\n");
           return -1;
        }
        printf("DEBUG (enviarTramaSupervisao):%d bytes written\n", bytes);
          printf("0x%02X 0x%02X 0x%02X 0x%02X 0x%02X\n", frame[0], frame[1], frame[2],
frame[3], frame[4]);
            estatisticas.tramasEnviadas++;   // Atualiza a contagem de tramas enviadas na
estatística
        // Wait until all bytes have been written to the serial port
        sleep(1);
        return 0;
    }*/

    void enviarTramaSupervisao(int fd, unsigned char address, unsigned char control) {
          unsigned char frame[5] = {FLAG, address, control, address ^ control, FLAG}; //
Criação da trama de controlo
        writeBytesSerialPort(frame, 5);
            printf("DEBUG (enviarTramaSupervisao): A enviar frame de controlo: 0x%X\n",
control);
            estatisticas.tramasEnviadas++;   // Atualiza a contagem de tramas enviadas na
estatística
    }

    // Atualiza as estatísticas com base no resultado de uma trama enviada
    void actualizarEstadisticasEnvio(int aceito) {
        if (aceito) {
              estatisticas.tramasAceitas++;          // Incrementa tramas aceitas se a trama foi
recebida corretamente
        } else {
            estatisticas.tramasRejeitadas++;     // Incrementa tramas rejeitadas se ocorreu um
erro na receção
        }
    }

    // Incrementa a contagem de tramas recebidas
    void actualizarEstadisticasRecepcao() {
        estatisticas.tramasRecebidas++;
    }
```

```
// Exibe as estatísticas da conexão
void mostrarEstatisticas() {
    printf("=== Estatísticas da Conexão ===\n");
    printf("Tramas Enviadas: %d\n", estatisticas.tramasEnviadas);
    printf("Tramas Recebidas: %d\n", estatisticas.tramasRecebidas);
    printf("Tramas Rejeitadas: %d\n", estatisticas.tramasRejeitadas);
    printf("Tramas Aceitas: %d\n", estatisticas.tramasAceitas);
     printf("Total de bytes transmitidos: %d bytes\n", estatisticas.totalBytesTransmitidos);
    printf("Tempo total de transmissão: %.2f ms\n", estatisticas.tiempoTransmision);
    printf("Tempo total de receção: %.2f ms\n", estatisticas.tiempoRecepcion);
    printf("Tempo total de desconexão: %.2f ms\n", estatisticas.tiempoDesconexion);
    printf("================================\n");
}

// Função para calcular o CRC (verificação de redundância cíclica) de um buffer de dados
/*unsigned char calculateCRC(const unsigned char *buf, int bufSize) {
    unsigned char crc = 0;
    for (int i = 0; i < bufSize; i++) {
        crc ^= buf[i];
        for (int j = 0; j < 8; j++) {
            if (crc & 0x80) {
                crc = (crc << 1) ^ 0x07;
            } else {
                crc <<= 1;
            }
        }
    }
    return crc;
}*/

// Función para calcular el BCC2 como un XOR simple de los datos
unsigned char calculateBCC2(const unsigned char *buf, int bufSize) {
    unsigned char bcc2 = 0;
    for (int i = 0; i < bufSize; i++) {
        bcc2 ^= buf[i];
    }
    return bcc2;
}

// Simula un error en los datos con una probabilidad dada
/*int introduceError(float probability) {
    return ((float)rand() / RAND_MAX) < probability;
```

```
}*/

////////////////////////////////////////////////
// LLOPEN - Abre a conexão serial
////////////////////////////////////////////////
// Parâmetros: estrutura com os parâmetros de conexão
// Retorna: o descritor da porta serial se bem-sucedido, -1 caso contrário
int llopen(LinkLayer connectionParameters) {
    conexionStart = clock();          // Inicia o temporizador global da conexão
                                fd      =      openSerialPort(connectionParameters.serialPort,
connectionParameters.baudRate);
    currentRole = connectionParameters.role;

    // Verifica se a porta serial foi aberta corretamente
    if (fd < 0) {
        fprintf(stderr, "Error al abrir la conexión serial\n");
        return -1;
    }

        timeout = connectionParameters.timeout;          // Define o tempo limite para
retransmissão
     retransmissions = connectionParameters.nRetransmissions;    // Define o número de
retransmissões permitidas
    int attempt_count = 0;
    clock_t start = clock();    // Inicia o temporizador para monitorar o tempo de conexão
                printf("DEBUG (llopen): Iniciando conexión en modo %s\n",
connectionParameters.role == LITx ? "Transmisor" : "Receptor");
    LinkLayerState state = START;
    (void)signal(SIGALRM, alarmHandler);

    switch (connectionParameters.role) {

        // Caso o rol seja de Transmissor
        case LITx:{
            while (retransmissions > 0) {
                unsigned char supFrame[5] = {FLAG, Address_Transmitter, Command_SET,
Address_Transmitter ^ Command_SET, FLAG};
                write(fd, supFrame, 5);
                alarm(timeout);
                alarmEnabled = 0;
                while (state != STOP_R  && !alarmEnabled) {

                    unsigned char byte = 0;
```

```
                int bytes;

                if((bytes = readByteSerialPort(&byte)) < 0){
                    printf("DEBUG (llopen Tx): Error receiving UA\n");
                    return -1;
                }

                if(bytes){

                    switch (state) {
                        case START:
                            if (byte == FLAG) state = FLAG_RCV;
                            break;
                        case FLAG_RCV:
                            if (byte == Address_Receiver) state = A_RCV;
                            else if (byte == FLAG) state = FLAG_RCV;
                            else state = START;  // Reseta caso o byte não seja esperado
                            break;
                        case A_RCV:
                            if (byte == Command_UA) state = C_RCV;
                            else if (byte == FLAG) state = FLAG_RCV;
                            else state = START;  // Reseta caso o byte não seja esperado
                            break;
                        case C_RCV:
                            if (byte == (Address_Receiver ^ Command_UA)) state = BCC1_OK;
                            else if (byte == FLAG) state = FLAG_RCV;
                            else state = START;
                            break;
                        case BCC1_OK:
                            if (byte == FLAG) state = STOP_R;
                            else state = START;
                            break;
                        default:
                            state = START;
                            break;

                    }
                }
            }
            // Se a conexão foi estabelecida (estado STOP_R alcançado)
            if (state == STOP_R) {
                        estatisticas.tiempoTransmision += (double)(clock() - start) /
CLOCKS_PER_SEC;
```

```
                printf("DEBUG (llopen Tx): Conexión establecida correctamente.\n");
                return fd;
            }

            // Se o alarme foi acionado (timeout)
            if (alarmEnabled) {
                desconexionStart = clock();
                    estatisticas.tiempoDesconexion += (double)(clock() - desconexionStart) *
1000.0 / CLOCKS_PER_SEC;
            }
            retransmissions--;
            printf("DEBUG (llopen Tx): Reintento restante = %d\n", retransmissions);
        }
        // Caso não seja possível estabelecer a conexão após todas as tentativas
        printf("DEBUG (llopen Tx): Error, no se pudo establecer la conexión.\n");
        return -1;
    }

    // Caso o rol seja de Receptor
    case LlRx:{

        while(state != STOP_R) {
            unsigned char byte;
            int bytes;
            if((bytes = readByteSerialPort(&byte)) < 0){
                attempt_count++;
                printf("DEBUG (llopen Rx): Error receiving UA\n");
                return -1;
            }
            if(bytes > 0) {
                // Lê bytes da porta serial e processa-os de acordo com o estado atual
                switch (state) {
                    case START:
                        if (byte == FLAG) state = FLAG_RCV;
                        else state = START;
                        break;
                    case FLAG_RCV:
                        if (byte == Address_Transmitter) state = A_RCV;
                        else if (byte == FLAG) state = FLAG_RCV;
                        else state = START;
                        break;
                    case A_RCV:
                        if (byte == Command_SET) state = C_RCV;
```

```
                    else if (byte == FLAG) state = FLAG_RCV;
                    else state = START;
                    break;
                case C_RCV:
                        if (byte == (Address_Transmitter ^ Command_SET)) state =
BCC1_OK;
                    else state = START;
                    break;
                case BCC1_OK:
                    if (byte == FLAG) state = STOP_R;
                    else state = START;
                    break;
                default:
                    state = START;
                    break;
            }
        }
    }
        unsigned char uaFrame[5] = {FLAG, Address_Receiver, Command_UA,
Address_Receiver ^ Command_UA, FLAG};
    write(fd, uaFrame, 5);

    // Registra o tempo de recepção
        estatisticas.tiempoRecepcion += (double)(clock() - start) * 1000.0 /
CLOCKS_PER_SEC;
    printf("DEBUG (llopen Rx): Conexión establecida y UA enviado.\n");
    return fd;
  }
  // Retorna erro se o rol não for válido
  default:
    break;
 }
 return -1;
}

// Realiza o byte stuffing nos dados de entrada
int applyByteStuffing(const unsigned char *input, int length, unsigned char *output) {
 int stuffedIndex = 0;       // Índice para o array de saída

 // Percorre cada byte do array de entrada
 for (int i = 0; i < length; i++) {
  printf("DEBUG (applyByteStuffing): Byte original = 0x%X\n", input[i]);
```

```
            // Verifica se o byte atual é um FLAG
            if (input[i] == FLAG) {
                output[stuffedIndex++] = ESCAPE;
                output[stuffedIndex++] = 0x5E;
                        printf("DEBUG (applyByteStuffing): FLAG detectado, aplicando stuffing ->
ESCAPE + 0x5E\n");
            }
            // Verifica se o byte atual é um ESCAPE
            else if (input[i] == ESCAPE) {
                output[stuffedIndex++] = ESCAPE;
                output[stuffedIndex++] = 0x5D;
                    printf("DEBUG (applyByteStuffing): ESCAPE detectado, aplicando stuffing ->
ESCAPE + 0x5D\n");
            }
                // Caso não seja FLAG nem ESCAPE, copia o byte para o array de saída sem
alteração
            else {
                output[stuffedIndex++] = input[i];
                printf("DEBUG (applyByteStuffing): Byte sem alteração = 0x%X\n", input[i]);
            }
        }
        printf("DEBUG (applyByteStuffing): Tamanho final após stuffing = %d\n", stuffedIndex);
        // Retorna o tamanho do array de saída após stuffing
        return stuffedIndex;
    }

    ////////////////////////////////////////////////
    // LLWRITE - Envia uma trama de dados
    ////////////////////////////////////////////////
    // Parâmetros:
    //   buf: ponteiro para o buffer que contém os dados a serem enviados
    //   bufSize: tamanho do buffer de dados
    // Retorna:
    //   0 se a trama for enviada com sucesso e confirmada, -1 em caso de erro

    int llwrite(const unsigned char *buf, int bufSize) {
        if (estatisticas.tiempoTransferencia == 0) {
            estatisticas.tiempoTransferencia = (double)clock();
        }

        unsigned char frame[MAX_FRAME_SIZE];
        int frameIndex = 0;
```

```
frame[frameIndex++] = FLAG;
frame[frameIndex++] = Address_Transmitter;
frame[frameIndex++] = Command_DATA;
frame[frameIndex++] = Address_Transmitter ^ Command_DATA;

unsigned char BCC2 = calculateBCC2(buf, bufSize);
frameIndex += applyByteStuffing(buf, bufSize, &frame[frameIndex]);
frameIndex += applyByteStuffing(&BCC2, 1, &frame[frameIndex]);
frame[frameIndex++] = FLAG;

int tentativas = retransmissions;
while (tentativas > 0) {
    // Enviar la trama completa
    writeBytesSerialPort(frame, frameIndex);
    alarmEnabled = 0;
    alarm(timeout);
    estatisticas.tramasEnviadas++;

    unsigned char byte;
    LinkLayerState state = START;
    while (!alarmEnabled && state != STOP_R) {
        if (readByteSerialPort(&byte) > 0) {
            switch (state) {
                case START:
                    if (byte == FLAG) state = FLAG_RCV;
                    break;
                case FLAG_RCV:
                    if (byte == Address_Receiver) state = A_RCV;
                    break;
                case A_RCV:
                    if (byte == C_RR0 || byte == C_RR1) {
                        state = STOP_R;  // Confirmación de recepción correcta
                    } else if (byte == C_REJ0 || byte == C_REJ1) {
                        // Reiniciar la transmisión al recibir REJ
                        state = START;
                        printf("DEBUG (llwrite): REJ received, resending frame...\n");
                        break; // Salir del bucle interno para reenviar la trama completa
                    }
                    break;
                default:
                    state = START;
                    break;
            }
```

```
            }
        }

        // Si se recibió RR, confirmar y avanzar
        if (state == STOP_R && (byte == C_RR0 || byte == C_RR1)) {
            alarm(0);
            actualizarEstadisticasEnvio(1);
            estatisticas.totalBytesTransmitidos += bufSize;

            return frameIndex;  // Confirmación exitosa, avanza al siguiente paquete
        }

        // Si se recibió REJ, reducir el contador de intentos y reiniciar
        tentativas--;
        desconexionStart = clock();
          estatisticas.tiempoDesconexion += (double)(clock() - desconexionStart) * 1000.0 /
CLOCKS_PER_SEC;
    }

    // Si todos los intentos fallan, retorno con error
    actualizarEstadisticasEnvio(0);
    printf("DEBUG (llwrite): Error, no se pudo enviar la trama correctamente.\n");
    return -1;
}

//Esta função remove bytes de escape de uma trama recebida, restaurando os bytes
originais
int applyByteDestuffing(const unsigned char *input, int length, unsigned char *output) {

    int destuffedIndex = 0;     // Índice para o array de saída
    int escape = 0;             // Indicador que verifica se o byte ESCAPE foi encontrado

    for (int i = 0; i < length; i++) {

        // Se o byte ESCAPE já foi encontrado, verifica o próximo byte
        if (escape) {
            if (input[i] == 0x5E) {
                output[destuffedIndex++] = FLAG;
                 printf("DEBUG (applyByteDestuffing): ESCAPE seguido de 0x5E, convertendo
para FLAG\n");
            }
            else if (input[i] == 0x5D) {
                output[destuffedIndex++] = ESCAPE;
```

```
                printf("DEBUG (applyByteDestuffing): ESCAPE seguido de 0x5D, convertendo para ESCAPE\n");
            }
            escape = 0;     // Reinicia o indicador de ESCAPE
        }
        else if (input[i] == ESCAPE) {
            escape = 1;     // Marca que o byte ESCAPE foi encontrado e aguarda o próximo byte
            printf("DEBUG (applyByteDestuffing): Byte ESCAPE detectado, aguardando próximo byte\n");
        }
        // Deteta o FLAG final e para o processamento
        else if (input[i] == FLAG && i == length - 1) {
            printf("DEBUG (applyByteDestuffing): FLAG final detectado, parando processamento\n");
            break;
        }
        // Copia o byte sem alteração se não houver destuffing
        else {
            output[destuffedIndex++] = input[i];
            printf("DEBUG (applyByteDestuffing): Byte sem alteração = 0x%X\n", input[i]);
        }
    }
    printf("DEBUG (applyByteDestuffing): Tamanho final após destuffing = %d\n", destuffedIndex);
    // Retorna o tamanho final do array de saída após o destuffing
    return destuffedIndex;
}

////////////////////////////////////////////////
// LLREAD - Lê uma trama de dados
////////////////////////////////////////////////
// Parâmetros:
//   packet: ponteiro para o buffer onde os dados recebidos serão armazenados
// Retorna:
//     O tamanho do pacote de dados (sem FLAG, A, C, e BCC1) se for recebido corretamente,
//   -1 em caso de erro.
int llread(unsigned char *packet) {
    LinkLayerState state = START;
    unsigned char frame[MAX_FRAME_SIZE];
    int frameIndex = 0;
    unsigned char byte;
```

```
        int tentativas = retransmissions;

        // Loop principal de tentativas de leitura
        while (tentativas > 0) {
            alarmEnabled = 0;
            alarm(timeout);
                printf("DEBUG (llread): Inicio do loop de leitura, tentativas restantes = %d\n",
tentativas);

                // Loop para ler bytes enquanto o alarme não dispara e o estado final não é
alcançado
            while (!alarmEnabled && state != STOP_R) {
                if (readByteSerialPort(&byte) > 0) {
                    printf("DEBUG (llread): Estado = %d, Byte recebido = 0x%X\n", state, byte);
                    switch (state) {
                        case START:
                            if (byte == FLAG) {
                                state = FLAG_RCV;
                                printf("DEBUG (llread): Transição para FLAG_RCV\n");
                            }
                            break;
                        case FLAG_RCV:
                            if (byte == Address_Transmitter) {
                                state = A_RCV;
                                printf("DEBUG (llread): Transição para A_RCV\n");
                            }
                            break;
                        case A_RCV:
                            if (byte == Command_DATA) {
                                state = C_RCV;
                                printf("DEBUG (llread): Transição para C_RCV (Command_DATA)\n");
                            } else if (byte == Command_DISC) {
                                                printf("DEBUG (llread): Command_DISC recebido,
desconectando...\n");
                                return -2;
                            }
                            break;
                        case C_RCV:
                            if (byte == (Address_Transmitter ^ Command_DATA)) {
                                state = BCC1_OK;
                                printf("DEBUG (llread): BCC1 OK, transição para DATA\n");
                            }
                            break;
```

```
                case BCC1_OK:
                    if (byte != FLAG) {
                        frame[frameIndex++] = byte;
                        state = DATA;
                                printf("DEBUG (llread): Transição para DATA, dado recebido =
0x%X\n", byte);
                    }
                    break;
                case DATA:
                    if (byte == FLAG) {
                        state = STOP_R;
                                 printf("DEBUG (llread): FLAG de fim recebido, transição para
STOP_R\n");
                    } else {
                        frame[frameIndex++] = byte;
                        printf("DEBUG (llread): Dado adicionado ao frame = 0x%X\n", byte);
                    }
                    break;
                default:
                    state = START;
                                printf("DEBUG (llread): Estado desconhecido, reiniciando para
START\n");
                    break;
            }
        }
    }

    // Processa a trama recebida se o estado final for alcançado
    if (state == STOP_R) {
        int destuffedSize = applyByteDestuffing(frame, frameIndex, packet);
        unsigned char BCC2 = calculateBCC2(packet, destuffedSize - 1);
            printf("DEBUG (llread): Tamanho após destuffing = %d, BCC2 calculado =
0x%X\n", destuffedSize, BCC2);

        // Verifica o BCC2 para garantir a integridade dos dados
        if (BCC2 == packet[destuffedSize - 1]) {
            printf("DEBUG (llread): Trama recebida corretamente. A enviar RR...\n");
            if (tramaRx == 0) {
                enviarTramaSupervisao(fd, Address_Receiver, C_RR0);
            } else {
                enviarTramaSupervisao(fd, Address_Receiver, C_RR1);
            }
            tramaRx = (tramaRx + 1) % 2;
```

```
                actualizarEstadisticasRecepcao();
                estatisticas.tramasRecebidas++;
                return destuffedSize - 1;
            } else {
                // Envia REJ se o BCC2 for incorreto
                printf("DEBUG (llread): Erro: BCC2 incorreto. A enviar REJ...\n");
                if (tramaRx == 0) {
                    enviarTramaSupervisao(fd, Address_Receiver, C_REJ0);
                } else {
                    enviarTramaSupervisao(fd, Address_Receiver, C_REJ1);
                }
                tentativas--;
                state = START;
                frameIndex = 0;
                    printf("DEBUG (llread): Reinicio após REJ, tentativas restantes = %d\n",
tentativas);
            }
        } else if (alarmEnabled) {
            // Control del tiempo de espera, registra y reinicia
            printf("DEBUG (llread): Tiempo de espera agotado, reintentando...\n");
            desconexionStart = clock();
             estatisticas.tiempoDesconexion += (double)(clock() - desconexionStart) * 1000.0
/ CLOCKS_PER_SEC;
            tentativas--;
            state = START;  // Reinicia el estado en caso de timeout
            frameIndex = 0; // Reinicia el índice del frame
                printf("DEBUG (llread): Reinicio após timeout, tentativas restantes = %d\n",
tentativas);
        }
    }

    printf("DEBUG (llread): Error, no se pudo recibir la trama correctamente.\n");
    return -1;

}

////////////////////////////////////////////////
// LLCLOSE - Fecha a conexão
////////////////////////////////////////////////
// Parâmetros:
//   showStatistics: indica se as estatísticas devem ser mostradas após o fechamento da
conexão
// Retorna:
```

```
//   0 se a conexão for fechada corretamente, -1 em caso de erro.
int llclose(int showStatistics) {
    printf("DEBUG: Iniciando función llclose.\n");
    LinkLayerState state = START;
    clock_t start = clock();
    // Se o rol atual é de Transmissor (LlTx)
    if (currentRole == LlTx) {
        // Envia a trama DISC para iniciar a desconexão
        enviarTramaSupervisao(fd, Address_Transmitter, Command_DISC);
        // Loop para tentar receber o DISC do receptor e confirmar o encerramento
        while (state != STOP_R && retransmissions > 0) {
            alarmEnabled = 0;
            alarm(timeout);

            unsigned char byte;
            while (!alarmEnabled && state != STOP_R) {
                if (readByteSerialPort(&byte) > 0) {
                    // Máquina de estados para verificar e processar o DISC do receptor
                    switch (state) {
                        case START:
                            if (byte == FLAG) state = FLAG_RCV;
                            break;
                        case FLAG_RCV:
                            if (byte == Address_Receiver) state = A_RCV;
                            break;
                        case A_RCV:
                            if (byte == Command_DISC) state = C_RCV;
                            break;
                        case C_RCV:
                            if (byte == (Address_Receiver ^ Command_DISC)) state = BCC1_OK;
                            break;
                        case BCC1_OK:
                            if (byte == FLAG) state = STOP_R;
                            break;
                        default:
                            break;
                    }
                }
            }

        }
        // Envia a trama de confirmação UA após receber DISC do receptor
        enviarTramaSupervisao(fd, Address_Transmitter, Command_UA);
```

```
            estatisticas.tiempoTransmision += (double)(clock() - start) / CLOCKS_PER_SEC;
        }
        // Caso o rol seja Receptor (LlRx)
        else if (currentRole == LlRx) {
            while (state != STOP_R) {
                // Loop para tentar receber o DISC do transmissor
                unsigned char byte;
                if (readByteSerialPort(&byte) > 0) {
                    // Máquina de estados para processar o DISC do transmissor
                    switch (state) {
                        case START:
                            if (byte == FLAG) state = FLAG_RCV;
                            break;
                        case FLAG_RCV:
                            if (byte == Address_Transmitter) state = A_RCV;
                            break;
                        case A_RCV:
                            if (byte == Command_DISC) state = C_RCV;
                            break;
                        case C_RCV:
                            if (byte == (Address_Transmitter ^ Command_DISC)) state = BCC1_OK;
                            break;
                        case BCC1_OK:
                            if (byte == FLAG) state = STOP_R;
                            break;
                        default:
                            break;
                    }
                }
            }
            // Envia o DISC ao transmissor para confirmar a desconexão
            enviarTramaSupervisao(fd, Address_Receiver, Command_DISC);
            estatisticas.tiempoRecepcion += (double)(clock() - start) / CLOCKS_PER_SEC;
        }
         estatisticas.tiempoTransferencia = ((double)clock() - estatisticas.tiempoTransferencia)
/ CLOCKS_PER_SEC;



        // Calcula a eficiência e exibe estatísticas, se solicitado

        int C = 9600; // Capacidade do enlace em bits por segundo
        int R = estatisticas.totalBytesTransmitidos * 8; // Total de bits transmitidos
```

```
        double eficiencia = ((double)R / (estatisticas.tiempoTransferencia * C)) * 100;

        // Exibe as estatísticas se showStatistics estiver ativo
        if (showStatistics) {
            mostrarEstatisticas();
                double tiempoTotalConexion = (double)(clock() - conexionStart) * 1000.0 /
CLOCKS_PER_SEC;
            printf("BaudRate: %d\n", C);
            printf("Tempo total da conexão: %.2f ms\n", tiempoTotalConexion);
                                printf("Tempo total de transferência: %.2f segundos\n",
estatisticas.tiempoTransferencia);
            printf("Total de bits transmitidos (R): %d bits\n", R);
            printf("Eficiência do protocolo (S): %.2f%%\n", eficiencia);
        }
         // Fecha a porta serial e retorna sucesso
        closeSerialPort();
        return 0;

    }
```