

**Protocolo de Ligação de Dados**

1º Trabalho Laboratorial

2023/2024

**Redes de Computadores**

**Turma 12 – Grupo 9**

- Francisco Dias Pires Ferreira de Sousa ([up202108715@fe.up.pt](mailto:up202005700@fe.up.pt))

- Simão Queirós Rodrigues ([up202005700@fe.up.pt](mailto:up202005700@fe.up.pt))

Porto, 8 de Novembro de 2023

**Sumário**

No contexto da Unidade Curricular de Redes de Computadores 2023/2024, o presente projeto focou-se na criação de um protocolo de comunicações de dados específico para a transferência de arquivos, recorrendo ao uso da Porta Série RS-232.

Com esta iniciativa, foi possível transpor para a prática os conhecimentos teóricos abordados durante as aulas teóricas, efetuando a implementação prática do protocolo em causa e reforçando a nossa compreensão sobre o mecanismo Stop-and-Wait.

**Introdução**

Este documento descreve o processo de conceção e avaliação de um protocolo de conexão de dados, seguindo as diretrizes estabelecidas no roteiro fornecido, para possibilitar a transferência de um arquivo através da porta série. O relatório está organizado em oito partes principais:  
  
 - Arquitetura: Descrição dos componentes funcionais e das interfaces empregadas.

- Estrutura do Código: Detalhes sobre as principais APIs, estruturas de dados e funções implementadas.

- Casos de uso principais: Exploração das operações chave do projeto, incluindo a sequência de invocação das funções.

- Protocolo de Ligação Lógica: Análise do funcionamento da conexão lógica e das táticas adotadas na sua implementação.

- Protocolo de Aplicação: Exame do funcionamento da camada de aplicação e das estratégias utilizadas na sua execução.

- Validação: Realização de testes para verificar a corretude da implementação.

- Eficiência do protocolo de ligação de dados: Não realizado.

- Conclusões: Consolidação das observações e resultados discutidos previamente.

**Arquitetura**

Blocos Funcionais

Este projeto foi estruturado em torno de duas camadas principais: a Camada de Ligação de Dados (LinkLayer) e a Camada de Aplicação (ApplicationLayer).

A Camada de Ligação de Dados é composta pela funcionalidade do protocolo mencionado anteriormente, contida nos arquivos link\_layer.h e link\_layer.c. Esta camada tem a função de gerir o início e o término das conexões, criação e envio de tramas de dados por meio da porta série, além de verificar a integridade das tramas recebidas e gerar sinais de erro se ocorrerem falhas na transmissão.

Por outro lado, a Camada de Aplicação, presente nos arquivos application\_layer.h e application\_layer.c, recorre à API da LinkLayer para efetuar a transferência e a receção de pacotes de dados que compõem um arquivo. Esta camada é aquela que opera mais próxima do usuário, permitindo definir parâmetros como o tamanho das tramas de informação, a velocidade de transferência de dados e o limite máximo de retransmissões permitidas.

Interfaces

A interação com o programa ocorre por meio de duas interfaces de comando, uma em cada computador, configuradas de maneira que uma aja como transmissor executando o binário e o outro como receptor.

**Estrutura do código**

ApplicationLayer

A screen shot of a computer code

Description automatically generated

LinkLayer

A computer screen shot of text

Description automatically generatedA computer screen shot of text

Description automatically generatedA computer screen shot of text

Description automatically generated

A screen shot of a computer program

Description automatically generated

**Casos de uso principais**

Neste documento, abordamos a implementação de um protocolo de comunicação para a transferência de ficheiros, utilizando a porta série RS-232. A execução do programa é adaptável, podendo operar tanto em modo transmissor como recetor. Cada modo emprega um conjunto específico de funções e uma sequência única de operações.

No Modo Transmissor:

-Estabelecimento da Ligação: A função **`llopen()`** inicia o processo, realizando o handshake inicial entre transmissor e recetor. Esta etapa envolve a troca de pacotes de controlo e a conexão com a porta série, através da função `connection()`.

- Preparação dos Dados: Utiliza-se **`getData()`** para obter o conteúdo do ficheiro que será enviado.

- Criação do Pacote de Controlo: Emprega-se **`getControlPacket()`** para formar um pacote de controlo inicial.

- Envio de Tramas: A função **`llwrite()`** é responsável por compor e enviar tramas de informação através da porta série, utilizando os dados fornecidos.

- Validação da Resposta do Recetor: Após o envio, a função **`readControlFrame()`** entra em ação como uma máquina de estados, lendo e validando as respostas do recetor.

- Terminação da Ligação: Por fim, a função **`llclose()`** é usada para finalizar a ligação, concluindo a troca de pacotes de controlo.

No Modo Recetor:

- Gestão de Tramas Recebidas: Aqui, a função **`llread()`** atua como uma máquina de estados, gerindo e validando tanto as tramas de controlo como as tramas de dados recebidas.

- Envio de Trama de Supervisão: A função **`sendSupervisionFrame()`** cria e envia tramas de supervisão pela porta série, baseando-se nas tramas lidas por `llread()`.

- Análise do Pacote de Controlo: **`parseControlPacket()`** é usada para extrair informações do ficheiro a ser transferido, que estão contidas no pacote de controlo no formato TLV.

- Obtenção do Segmento de Dados: Por fim, a função **`getDataPacket()`** retorna um segmento específico do ficheiro, conforme definido no pacote de dados recebido.

**Protocolo de Ligação Lógica**

A camada de ligação de dados, na nossa implementação, atua diretamente com a Porta Série e desempenha um papel crucial na comunicação entre o emissor e o recetor. Adotamos o protocolo Stop-and-wait tanto para estabelecer e terminar a ligação, como para o envio de tramas de supervisão e de informação.

O processo de estabelecimento da ligação começa com a função **`llopen`**. Depois de configurada a porta série, o emissor envia uma trama de supervisão SET, aguardando pela resposta do recetor com uma trama UA. Quando o recetor recebe o SET, responde com UA. Se o emissor receber corretamente a trama UA, a ligação considera-se estabelecida com sucesso. Após este passo, o emissor inicia o envio de informações para serem lidas pelo recetor.

O envio destas informações é realizado através da função **`llwrite`**. Esta função processa um pacote de controlo ou de dados, aplicando a técnica de byte stuffing para evitar conflitos com bytes que possam ser idênticos às flags da trama. Em seguida, transforma este pacote numa trama de informação e envia-a para o recetor, ficando à espera da sua resposta. Se a trama for rejeitada, o processo repete-se até ser aceite ou até que se exceda o número máximo de tentativas. Cada tentativa de envio é limitada por um tempo específico, após o qual se considera um time-out.

A receção e leitura de informação são executadas pela função **`llread`**. Esta função processa a informação recebida pela porta série e verifica a sua validade. Primeiramente, realiza o processo inverso do byte stuffing no campo de dados da trama e valida os códigos de verificação de erro BCC1 e BCC2, assegurando que não ocorreram erros durante a transmissão.

Finalmente, a ligação é terminada através da função **`llclose`**. Esta função é acionada pelo emissor, quer quando se atinge o limite de tentativas fracassadas, quer quando a transferência de pacotes de dados está completa. O emissor envia então uma trama de supervisão DISC e aguarda por uma resposta idêntica do recetor, finalizando assim a sua operação. Quando o emissor recebe novamente um DISC, responde com UA e encerra a ligação.

**Protocolo de Aplicação**

Na nossa abordagem, a camada de aplicação assume um papel interativo com o ficheiro a ser transferido e com o utilizador. Esta camada permite a definição de vários parâmetros cruciais, como o ficheiro a ser transferido, a porta série a utilizar, a velocidade da transferência, o tamanho dos bytes de dados de cada pacote, o limite de retransmissões permitidas e o tempo máximo de espera por uma resposta do recetor. A transferência do ficheiro é efetuada através da utilização da API da LinkLayer, que converte os pacotes de dados em tramas de informação.

Quando o processo de handshake entre transmissor e recetor está concluído, o conteúdo completo do ficheiro é copiado para um buffer local usando a função **`getData`**. Este conteúdo é então fragmentado pela camada applicationLayer de acordo com o número de bytes especificados. O primeiro pacote a ser enviado pelo transmissor contém dados no formato TLV (Type, Length, Value), gerado pela função **`getControlPacket`**. Este pacote informa o tamanho do ficheiro e o seu nome. Do lado do recetor, este pacote é processado pela função **`parseControlPacket`**, que se encarrega de criar e alocar o espaço necessário para a receção do ficheiro.

Cada segmento do ficheiro a ser transferido é embutido num pacote de dados pela função **`getDataPacket`** e enviado através da porta série utilizando a função `llwrite` da API. Cada envio é seguido de uma resposta do recetor, que pode aceitar ou rejeitar o pacote. Se o pacote for aceite, o transmissor procede com o envio do fragmento seguinte; se for rejeitado, reenvia o mesmo fragmento. O recetor avalia cada pacote individualmente através das funções `llread` e **`parseDataPacket`**, extraindo do pacote o segmento original do ficheiro quando este é recebido corretamente.

A conexão entre as duas máquinas é finalizada com a invocação da função `llclose` da API, que ocorre após a conclusão da transferência dos pacotes de dados ou caso se exceda o número máximo de tentativas permitidas.

**Validação**

Para assegurar a eficácia e a correta implementação do protocolo que desenvolvemos realizámos uma série de testes específicos, focados na resiliência e na confiabilidade do protocolo Stop-And-Wait que implementámos:

- **Interrupção Parcial ou Total da Porta Série:** Testámos a capacidade do protocolo de lidar com interrupções na comunicação, seja uma interrupção completa ou apenas parcial. Este teste ajudou a verificar a robustez do protocolo perante falhas na conexão.

**- Introdução de Ruído na Porta Série Através de Curto-Circuito:** Para testar a resistência do protocolo a interferências externas, introduzimos deliberadamente ruído na Porta Série, simulando condições adversas de transmissão.

Estes testes foram reproduzidos na presença do docente durante a apresentação do projeto numa aula laboratorial.

**Eficiência do protocolo de ligação de dados**

Não realizado

**Conclusões**

A implementação do protocolo de ligação de dados, composto pela LinkLayer, responsável pela interação com a porta série e gestão das tramas de informação, e pela ApplicationLayer, que tratou da interação direta com o ficheiro a ser transferido, revelou-se fundamental para a aplicação prática e compreensão dos conceitos teóricos abordados nas aulas. Através deste projeto, conseguimos assimilar efetivamente técnicas como o byte stuffing e framing, além de aprofundar o nosso entendimento sobre o funcionamento do protocolo Stop-and-Wait, particularmente no que toca à deteção e gestão de erros.

**Anexo I – application\_layer.h**

#ifndef \_APPLICATION\_LAYER\_H\_

#define \_APPLICATION\_LAYER\_H\_

#include <stdio.h>

// Inicia a transferência ou recepção de um arquivo pela porta série

void applicationLayer(const char \*serialPort, const char \*role, int baudRate,

                      int nTries, int timeout, const char \*filename);

// Extrai o nome e o tamanho do arquivo de um pacote de controle recebido

unsigned char\* parseControlPacket(unsigned char\* stream, int streamLength, unsigned long int \*outputFileSize);

// Processa um pacote de dados recebido

void parseDataPacket(const unsigned char\* packet, const unsigned int packetSize, unsigned char\* buffer);

// Cria um pacote de controle com informações sobre o arquivo a ser transferido

unsigned char \* getControlPacket(unsigned int type, const char \*fileIdentifier, long fileSize, unsigned int \*frameSize);

// Monta um pacote de dados com um segmento do arquivo a ser transmitido e seu número de sequência

unsigned char \* getDataPacket(unsigned char seqNumber, const unsigned char \*payload, int payloadSize, int \*frameLength);

// Lê o conteúdo do arquivo para a memória

unsigned char \* getData(FILE\* fd, long int fileLength);

#endif // \_APPLICATION\_LAYER\_H\_

**Anexo II - application\_layer.c**

#include "application\_layer.h"

#include "link\_layer.h"

#include <string.h>

#include <fcntl.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <termios.h>

#include <unistd.h>

#include <math.h>

void applicationLayer(const char \*serialPort, const char \*role, int baudRate,

                      int nTries, int timeout, const char \*filename)

{

    LinkLayer linkLayer;

    strcpy(linkLayer.serialPort,serialPort);

    linkLayer.role = strcmp(role, "tx") ? LlRx : LlTx;

    linkLayer.baudRate = baudRate;

    linkLayer.nRetransmissions = nTries;

    linkLayer.timeout = timeout;

    int fd = llopen(linkLayer);

    if (fd < 0) {

        perror("Connection error\n");

        exit(-1);

    }

    switch (linkLayer.role) {

        case LlTx: {

            FILE\* file = fopen(filename, "rb");

            if (file == NULL) {

                perror("File not found\n");

                exit(-1);

            }

            int prev = ftell(file);

            fseek(file,0L,SEEK\_END);

            long int fileSize = ftell(file)-prev;

            fseek(file,prev,SEEK\_SET);

            unsigned int cpSize;

            unsigned char \*controlPacketStart = getControlPacket(2, filename, fileSize, &cpSize);

            if(llwrite(fd, controlPacketStart, cpSize) == -1){

                printf("Exit: error in start packet\n");

                exit(-1);

            }

            unsigned char sequence = 0;

            unsigned char\* content = getData(file, fileSize);

            long int bytesLeft = fileSize;

            while (bytesLeft >= 0) {

                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;

                unsigned char\* packet = getDataPacket(sequence, data, dataSize, &packetSize);

                if(llwrite(fd, packet, packetSize) == -1) {

                    printf("Exit: error in data packets\n");

                    exit(-1);

                }

                bytesLeft -= (long int) MAX\_PAYLOAD\_SIZE;

                content += dataSize;

                sequence = (sequence + 1) % 255;

            }

            unsigned char \*controlPacketEnd = getControlPacket(3, filename, fileSize, &cpSize);

            if(llwrite(fd, controlPacketEnd, cpSize) == -1) {

                printf("Exit: error in end packet\n");

                exit(-1);

            }

            llclose(fd);

            break;

        }

        case LlRx: {

            unsigned char \*packet = (unsigned char \*)malloc(MAX\_PAYLOAD\_SIZE);

            int packetSize = -1;

            while ((packetSize = llread(fd, packet)) < 0);

            unsigned long int rxFileSize = 0;

            unsigned char\* name = parseControlPacket(packet, packetSize, &rxFileSize);

            FILE\* newFile = fopen((char \*) name, "wb+");

            while (1) {

                while ((packetSize = llread(fd, packet)) < 0);

                if(packetSize == 0) break;

                else if(packet[0] != 3){

                    unsigned char \*buffer = (unsigned char\*)malloc(packetSize);

                    parseDataPacket(packet, packetSize, buffer);

                    fwrite(buffer, sizeof(unsigned char), packetSize-4, newFile);

                    free(buffer);

                }

                else continue;

            }

            fclose(newFile);

            break;

        default:

            exit(-1);

            break;

    }}

}

unsigned char\* parseControlPacket(unsigned char\* stream, int streamLength, unsigned long int \*outputFileSize) {

    // Extracting the file size from the packet

    unsigned char sizeDescriptor = stream[2]; // size of the file size field

    unsigned char fileSizeBytes[sizeDescriptor];

    memcpy(fileSizeBytes, stream + 3, sizeDescriptor);

    \*outputFileSize = 0; // Ensure the file size starts at 0

    for(int index = 0; index < sizeDescriptor; ++index) {

        \*outputFileSize |= (unsigned long)(fileSizeBytes[sizeDescriptor - index - 1]) << (index \* 8);

    }

    // Extracting the file name from the packet

    unsigned char nameLength = stream[3 + sizeDescriptor + 1]; // size of the file name field

    unsigned char \*fileName = (unsigned char\*)malloc(nameLength + 1); // +1 for the null-terminator

    memcpy(fileName, stream + 3 + sizeDescriptor + 2, nameLength);

    fileName[nameLength] = '\0'; // Null-terminate the file name

    return fileName; // Return the extracted file name

}

void parseDataPacket(const unsigned char\* packet, const unsigned int packetSize, unsigned char\* buffer) {

    memcpy(buffer,packet+4,packetSize-4);

    buffer += packetSize+4;

}

unsigned char \* getControlPacket(unsigned int type, const char \*fileIdentifier, long fileSize, unsigned int \*frameSize) {

    // Determine the size of the file size field based on the value

    int fileSizeFieldLength = (int) ceil(log2(fileSize + 1) / 8);

    int fileIdentifierLength = strlen(fileIdentifier);

    \*frameSize = 1 + 2 + fileSizeFieldLength + 2 + fileIdentifierLength; // Calculate total frame size

    unsigned char \*controlFrame = (unsigned char \*)malloc(\*frameSize);

    unsigned int position = 0;

    controlFrame[position++] = type; // Type of control frame

    controlFrame[position++] = 0; // Separator

    controlFrame[position++] = fileSizeFieldLength; // File size field length

    // Encode the file size into the frame

    for (int i = fileSizeFieldLength - 1; i >= 0; i--) {

        controlFrame[position++] = (fileSize >> (i \* 8)) & 0xFF;

    }

    // Add file identifier length and content

    controlFrame[position++] = 1; // Separator

    controlFrame[position++] = fileIdentifierLength;

    memcpy(controlFrame + position, fileIdentifier, fileIdentifierLength);

    return controlFrame;

}

unsigned char \* getDataPacket(unsigned char seqNumber, const unsigned char \*payload, int payloadSize, int \*frameLength) {

    \*frameLength = 4 + payloadSize; // 1 byte for type, 1 for sequence number, 2 for size, and the rest for payload

    unsigned char \*dataFrame = (unsigned char \*)malloc(\*frameLength);

    dataFrame[0] = 1; // Data frame type identifier

    dataFrame[1] = seqNumber;

    dataFrame[2] = (payloadSize >> 8) & 0xFF;

    dataFrame[3] = payloadSize & 0xFF;

    memcpy(dataFrame + 4, payload, payloadSize);

    return dataFrame;

}

unsigned char \* getData(FILE\* fd, long int fileLength) {

    unsigned char\* content = (unsigned char\*)malloc(sizeof(unsigned char) \* fileLength);

    fread(content, sizeof(unsigned char), fileLength, fd);

    return content;

}

**Anexo III – link\_layer.h**

#ifndef \_LINK\_LAYER\_H\_

#define \_LINK\_LAYER\_H\_

#include <fcntl.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <termios.h>

#include <unistd.h>

#include <signal.h>

#include <time.h>

#define \_POSIX\_SOURCE 1

#define BAUDRATE 38400

#define MAX\_PAYLOAD\_SIZE 1000

#define BUF\_SIZE 256

#define FALSE 0

#define TRUE 1

#define FLAG 0x7E

#define ESC 0x7D

#define A\_ER 0x03

#define A\_RE 0x01

#define C\_SET 0x03

#define C\_DISC 0x0B

#define C\_UA 0x07

#define C\_ACKNOWLEDGE(Nr) ((Nr << 7) | 0x05)

#define C\_REJECTION(Nr) ((Nr << 7) | 0x01)

#define C\_CONTROL(Ns) (Ns << 6)

typedef enum

{

   LlTx,

   LlRx,

} LinkLayerRole;

typedef enum {

    START,

    FLAG\_RCV,

    ADDRESS\_RCV,

    CONTROL\_RCV,

    BCC1\_OK,

    STOP\_R,

    DATA\_FOUND\_ESC,

    READING\_DATA,

    DISCONNECTED,

    BCC2\_OK

} LinkLayerState;

typedef struct

{

    char serialPort[50];

    LinkLayerRole role;

    int baudRate;

    int nRetransmissions;

    int timeout;

} LinkLayer;

// Estabelece a conexão da camada de ligação de dados

int llopen(LinkLayer connectionParameters);

// Envia um pacote de dados pela camada de ligação

int llwrite(int descriptor, const unsigned char \*dataBuffer, int bufferSize);

// Espera e recebe um pacote de controlo

unsigned char secondaryReceiver(int descriptor);

// Realiza a função do transmissor primário

int primaryTransmitter(int descriptor, unsigned char \*packet, int packetIdx);

// Lê e processa um pacote de dados

int llread(int fd, unsigned char \*packet);

// Fecha a conexão da camada de ligação de dados

int llclose(int fd);

// Processa um byte recebido

void processReceivedByte(LinkLayerState\* state, unsigned char byte);

// Realiza a configuração inicial da porta série

int connection(const char\* port);

// Gerencia o sinal de alarme durante a comunicação

void alarmHandler(int signal);

// Lê o controlo da porta série

unsigned char readControlFrame (int portDescriptor);

//  Envia a supervisão

int sendSupervisionFrame(int portDescriptor, unsigned char addressField, unsigned char controlField);

#endif // \_LINK\_LAYER\_H\_

**Anexo IV - link\_layer.c**

#include "link\_layer.h"

volatile int STOP = FALSE;

int alarmSignaled = FALSE;

int alarmCount = 0;

int timeout = 0;

int retransmitions = 0;

unsigned char tramaTx = 0;

unsigned char tramaRx = 1;

int llopen(LinkLayer connectionParams) {

    LinkLayerState currentState = START;

    int fileDescriptor = connection(connectionParams.serialPort);

    if (fileDescriptor < 0) return -1;

    unsigned char receivedByte;

    timeout = connectionParams.timeout;

    retransmitions = connectionParams.nRetransmissions;

    if (connectionParams.role == LlTx) {

        signal(SIGALRM, alarmHandler);

        while (connectionParams.nRetransmissions != 0 && currentState != STOP\_R) {

            sendSupervisionFrame(fileDescriptor, A\_ER, C\_SET);

            alarm(connectionParams.timeout);

            alarmSignaled = FALSE;

            while (!alarmSignaled && currentState != STOP\_R) {

                if (read(fileDescriptor, &receivedByte, 1) > 0) {

                    switch (currentState) {

                        case START:

                            currentState = (receivedByte == FLAG) ? FLAG\_RCV : START;

                            break;

                        case FLAG\_RCV:

                            currentState = (receivedByte == A\_RE) ? ADDRESS\_RCV :

                                           (receivedByte == FLAG) ? FLAG\_RCV : START;

                            break;

                        case ADDRESS\_RCV:

                            currentState = (receivedByte == C\_UA) ? CONTROL\_RCV :

                                           (receivedByte == FLAG) ? FLAG\_RCV : START;

                            break;

                        case CONTROL\_RCV:

                            currentState = (receivedByte == (A\_RE ^ C\_UA)) ? BCC1\_OK :

                                           (receivedByte == FLAG) ? FLAG\_RCV : START;

                            break;

                        case BCC1\_OK:

                            currentState = (receivedByte == FLAG) ? STOP\_R : START;

                            break;

                        default:

                            break;

                    }

                }

            }

            connectionParams.nRetransmissions--;

        }

        if (currentState != STOP\_R) return -1;

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

        while (currentState != STOP\_R) {

            if (read(fileDescriptor, &receivedByte, 1) > 0) {

                switch (currentState) {

                    case START:

                        currentState = (receivedByte == FLAG) ? FLAG\_RCV : START;

                        break;

                    case FLAG\_RCV:

                        currentState = (receivedByte == A\_ER) ? ADDRESS\_RCV :

                                       (receivedByte == FLAG) ? FLAG\_RCV : START;

                        break;

                    case ADDRESS\_RCV:

                        currentState = (receivedByte == C\_SET) ? CONTROL\_RCV :

                                       (receivedByte == FLAG) ? FLAG\_RCV : START;

                        break;

                    case CONTROL\_RCV:

                        currentState = (receivedByte == (A\_ER ^ C\_SET)) ? BCC1\_OK :

                                       (receivedByte == FLAG) ? FLAG\_RCV : START;

                        break;

                    case BCC1\_OK:

                        currentState = (receivedByte == FLAG) ? STOP\_R : START;

                        break;

                    default:

                        break;

                }

            }

        }

        sendSupervisionFrame(fileDescriptor, A\_RE, C\_UA);

    } else {

        return -1;

    }

    return fileDescriptor;

}

int llwrite(int descriptor, const unsigned char \*dataBuffer, int bufferSize) {

    int packetLength = 6 + bufferSize;

    unsigned char \*packet = (unsigned char \*) malloc(packetLength);

    packet[0] = FLAG;

    packet[1] = A\_ER;

    packet[2] = C\_CONTROL(tramaTx);

    packet[3] = packet[1] ^ packet[2];

    memcpy(packet + 4, dataBuffer, bufferSize);

    unsigned char bccCheck = dataBuffer[0];

    for (unsigned int i = 1; i < bufferSize; i++) {

        bccCheck ^= dataBuffer[i];

    }

    int packetIdx = 4;

    for (unsigned int i = 0; i < bufferSize; i++) {

        if (dataBuffer[i] == FLAG || dataBuffer[i] == ESC) {

            packet = realloc(packet, ++packetLength);

            packet[packetIdx++] = ESC;

        }

        packet[packetIdx++] = dataBuffer[i];

    }

    packet[packetIdx++] = bccCheck;

    packet[packetIdx++] = FLAG;

    int result = primaryTransmitter(descriptor, packet, packetIdx);

    free(packet);

    return result;

}

unsigned char secondaryReceiver(int descriptor) {

    while (alarmSignaled == FALSE) {

        unsigned char response = readControlFrame(descriptor);

        if (!response) {

            continue;

        }

        else if (response == C\_REJECTION(0) || response == C\_REJECTION(1) ||

                 response == C\_ACKNOWLEDGE(0) || response == C\_ACKNOWLEDGE(1)) {

            return response;

        }

    }

    return 0;

}

int primaryTransmitter(int descriptor, unsigned char \*packet, int packetIdx) {

    int currentAttempt = 0;

    int hasRejections = 0, isAccepted = 0;

    while (currentAttempt < retransmitions) {

        alarmSignaled = FALSE;

        alarm(timeout);

        hasRejections = 0;

        isAccepted = 0;

        write(descriptor, packet, packetIdx);

        unsigned char response = secondaryReceiver(descriptor);

        if (response == C\_REJECTION(0) || response == C\_REJECTION(1)) {

            hasRejections = 1;

        } else if (response == C\_ACKNOWLEDGE(0) || response == C\_ACKNOWLEDGE(1)) {

            isAccepted = 1;

            tramaTx = (tramaTx + 1) % 2;

        }

        if (isAccepted) break;

        currentAttempt++;

    }

    if (isAccepted) return packetIdx;

    else {

        llclose(descriptor);

        return -1;

    }

}

int llread(int fd, unsigned char \*dataPacket) {

    unsigned char currentByte, controlField;

    int dataIndex = 0;

    LinkLayerState currentState = START;

    while (currentState != STOP\_R) {

        if (read(fd, &currentByte, 1) > 0) {

            switch (currentState) {

                case START:

                    currentState = (currentByte == FLAG) ? FLAG\_RCV : START;

                    break;

                case FLAG\_RCV:

                    if (currentByte == A\_ER)

                        currentState = ADDRESS\_RCV;

                    else if (currentByte != FLAG)

                        currentState = START;

                    break;

                case ADDRESS\_RCV:

                    if (currentByte == C\_CONTROL(0) || currentByte == C\_CONTROL(1)) {

                        controlField = currentByte;

                        currentState = CONTROL\_RCV;

                    } else if (currentByte == FLAG) {

                        currentState = FLAG\_RCV;

                    } else if (currentByte == C\_DISC) {

                        sendSupervisionFrame(fd, A\_RE, C\_DISC);

                        return 0;

                    } else {

                        currentState = START;

                    }

                    break;

                case CONTROL\_RCV:

                    currentState = (currentByte == (A\_ER ^ controlField)) ? READING\_DATA :

                                   (currentByte == FLAG) ? FLAG\_RCV : START;

                    break;

                case READING\_DATA:

                    if (currentByte == ESC)

                        currentState = DATA\_FOUND\_ESC;

                    else if (currentByte == FLAG) {

                        unsigned char bcc2 = dataPacket[dataIndex - 1];

                        dataIndex--;

                        dataPacket[dataIndex] = '\0';

                        unsigned char checksum = dataPacket[0];

                        for (unsigned int j = 1; j < dataIndex; j++)

                            checksum ^= dataPacket[j];

                        if (bcc2 == checksum) {

                            currentState = STOP\_R;

                            sendSupervisionFrame(fd, A\_RE, C\_ACKNOWLEDGE(tramaRx));

                            tramaRx = (tramaRx + 1) % 2;

                            return dataIndex;

                        } else {

                            printf("Error: retransmission\n");

                            sendSupervisionFrame(fd, A\_RE, C\_REJECTION(tramaRx));

                            return -1;

                        }

                    } else {

                        dataPacket[dataIndex++] = currentByte;

                    }

                    break;

                case DATA\_FOUND\_ESC:

                    currentState = READING\_DATA;

                    dataPacket[dataIndex++] = (currentByte == ESC || currentByte == FLAG) ? currentByte : ESC ^ currentByte;

                    break;

                default:

                    break;

            }

        }

    }

    return -1;

}

int llclose(int fd) {

    LinkLayerState state = START;

    unsigned char byte;

    (void) signal(SIGALRM, alarmHandler);

    while (retransmitions != 0 && state != STOP\_R) {

        sendSupervisionFrame(fd, A\_ER, C\_DISC);

        alarm(timeout);

        alarmSignaled = FALSE;

        while (!alarmSignaled && state != STOP\_R) {

            int bytesRead = read(fd, &byte, 1);

            if (bytesRead > 0) {

                processReceivedByte(&state, byte);

            }

        }

        retransmitions--;

    }

    if (state != STOP\_R) return -1;

    sendSupervisionFrame(fd, A\_ER, C\_UA);

    return close(fd);

}

void processReceivedByte(LinkLayerState\* state, unsigned char byte) {

    switch (\*state) {

        case START:

            if (byte == FLAG) \*state = FLAG\_RCV;

            break;

        case FLAG\_RCV:

            \*state = (byte == A\_RE) ? ADDRESS\_RCV : (byte != FLAG) ? START : FLAG\_RCV;

            break;

        case ADDRESS\_RCV:

            if (byte == C\_DISC) \*state = CONTROL\_RCV;

            else if (byte == FLAG) \*state = FLAG\_RCV;

            else \*state = START;

            break;

        case CONTROL\_RCV:

            \*state = (byte == (A\_RE ^ C\_DISC)) ? BCC1\_OK :

                     (byte == FLAG) ? FLAG\_RCV : START;

            break;

        case BCC1\_OK:

            \*state = (byte == FLAG) ? STOP\_R : START;

            break;

        default:

            break;

    }

}

int connection(const char\* port) {

    int serialFd = open(port, O\_RDWR | O\_NOCTTY);

    if (serialFd < 0) {

        perror("Error opening serial port");

        exit(-1);

    }

    struct termios oldSettings, newSettings;

    // Get the current serial port settings

    if (tcgetattr(serialFd, &oldSettings) != 0) {

        perror("Failed to get serial port attributes");

        close(serialFd);

        exit(-1);

    }

    // Configure the new settings

    bzero(&newSettings, sizeof(newSettings));

    newSettings.c\_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;

    newSettings.c\_iflag = IGNPAR;

    newSettings.c\_oflag = 0;

    newSettings.c\_lflag = 0;

    newSettings.c\_cc[VTIME] = 0;   /\* inter-character timer unused \*/

    newSettings.c\_cc[VMIN] = 0;    /\* non-blocking read \*/

    // Clear the serial port buffer

    tcflush(serialFd, TCIOFLUSH);

    // Apply the new settings

    if (tcsetattr(serialFd, TCSANOW, &newSettings) != 0) {

        perror("Failed to set serial port attributes");

        close(serialFd);

        exit(-1);

    }

    return serialFd;

}

void alarmHandler(int signal) {

    alarmSignaled = TRUE;

    alarmCount++;

}

unsigned char readControlFrame(int portDescriptor) {

    unsigned char receivedByte, controlField = 0;

    LinkLayerState currentState = START;

    while (currentState != STOP\_R && !alarmSignaled) {

        if (read(portDescriptor, &receivedByte, 1) > 0) {

            switch (currentState) {

                case START:

                    currentState = (receivedByte == FLAG) ? FLAG\_RCV : START;

                    break;

                case FLAG\_RCV:

                    currentState = (receivedByte == A\_RE) ? ADDRESS\_RCV :

                                  (receivedByte != FLAG) ? START : FLAG\_RCV;

                    break;

                case ADDRESS\_RCV:

                    if (receivedByte == C\_ACKNOWLEDGE(0) || receivedByte == C\_ACKNOWLEDGE(1) ||

                        receivedByte == C\_REJECTION(0) || receivedByte == C\_REJECTION(1) ||

                        receivedByte == C\_DISC) {

                        currentState = CONTROL\_RCV;

                        controlField = receivedByte;

                    } else if (receivedByte == FLAG) {

                        currentState = FLAG\_RCV;

                    } else {

                        currentState = START;

                    }

                    break;

                case CONTROL\_RCV:

                    currentState = (receivedByte == (A\_RE ^ controlField)) ? BCC1\_OK :

                                  (receivedByte == FLAG) ? FLAG\_RCV : START;

                    break;

                case BCC1\_OK:

                    currentState = (receivedByte == FLAG) ? STOP\_R : START;

                    break;

                default:

                    break;

            }

        }

    }

    return controlField;

}

int sendSupervisionFrame(int portDescriptor, unsigned char addressField, unsigned char controlField) {

    unsigned char supervisionFrame[5] = {

        FLAG,

        addressField,

        controlField,

        addressField ^ controlField,

        FLAG

    };

    return write(portDescriptor, supervisionFrame, sizeof(supervisionFrame));

}