U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Redes de Computadores 2024/2025

Protocolo de Ligação de Dados

1º trabalho laboratorial

Maria João Vieira (up202204802@up.pt)
Rodrigo Martins (up202008868@up.pt)
Turma 5

Resumo

Este trabalho foi realizado no âmbito da Unidade Curricular Redes de Computadores, com o objetivo de desenvolver um protocolo de ligação de dados, implementando um transmissor e um recetor para a transferência de um ficheiro, através de portas série RS-232.

O projeto envolveu a implementação das camadas de aplicação e de ligação de dados, incluindo a utilização de byte-stuffing e do controlo de erros para garantir uma transmissão fiável de dados. As principais conclusões incluem a validação bem-sucedida da implementação com diferentes tipos de ficheiros, embora a interrupção da transmissão não tenha sido corretamente implementada.

Através do desenvolvimento deste sistema, consolidamos os conhecimentos adquiridos na teoria, permitindo uma compreensão mais aprofundada de técnicas como byte-stuffing e mecanismos de controlo de erros.

Introdução

O objetivo deste trabalho é o desenvolvimento de um protocolo de ligação de dados, estando este relatório dividido nas seguintes secções:

- **Arquitetura**: Blocos funcionais e interfaces.
- **Estrutura do Código**: APIs utilizadas, principais estruturas de dados implementadas e funções mais relevantes e como estas se interligam com a arquitetura do sistema.
- **Casos de Uso Principais:** Identificação, sequência de chamadas de função.
- Protocolo de ligação lógica: Análise dos aspectos funcionais do protocolo de ligação lógica, incluindo a sua implementação.
- **Protocolo de aplicação**: Descrição dos elementos chave do protocolo de aplicação, incluindo a sua implementação.
- **Validação**: Apresentação dos testes realizados para garantir o funcionamento correto do protocolo;
- **Eficiência do Protocolo de Ligação de Dados**: Caracterização estatística da eficiência do protocolo, realizada por meio de medições no código desenvolvido.
- **Conclusões**: Resumo dos principais resultados e reflexão sobre os objetivos de aprendizagem atingidos durante o desenvolvimento deste trabalho.

Arquitetura

O projeto foi desenvolvido através de duas estruturas diferentes: a Application Layer e a LinkLayer. Também tivemos acesso a um ficheiro SerialPort, que apesar de não podermos modificar, facilitou a comunicação.

A ApplicationLayer, representada pela application_layer.c e application_layer.h funciona como papel da camada de aplicação. A sua implementação permitiu-nos o controlo dos dados. Ela utiliza a API do LinkLayer para transmitir e receber pacotes de controlo e pacatos de dados.

A LinkLayer, onde se encontram os ficheiros link_layer.c e link_layer.h funciona como camada de ligação de dados. Ela tem como objetivo estabelecer uma ligação e o término de uma conexão, bem como variadas coisas, tal como a criação e envio de tramas, validar as tramas recebidas ou ainda trocar mensagens de sucesso durante a conexão, transmissão ou desconexão.

Estrutura do Código

- ApplicationLayer

Na implementação deste ficheiro utilizamos a seguinte função:

```
void applicationLayer(const char *serialPort, const char *role, int baudRate,
                      int nTries, int timeout, const char *filename)
```

- LinkLayer

Na implementação desta classe, utilizamos a enum LinkLayerRole, que permitiu distinguir o recetor do transmissor e a struct LinkLayer, onde irão ser guardados os parâmetros que utilizamos no programa.

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

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

As funções implementadas no bloco da Camada de Ligação (LinkLayer) são as seguintes :

```
// Criação do alarme
void alarmHandler(int signal)

// Abre e estabelece a conexão entre o recetor e o transmissor
int llopen(LinkLayer connectionParameters)

// Lida com o envio de tramas
int llwrite(const unsigned char *buf, int bufSize)

// Aplica stuffing aos dados para evitar conflitos de flag.
int stuff(unsigned char *stuffed, const unsigned char *helper, int size2)

// Remove stuffing dos dados recebidos.
int destuff(unsigned char *destuffed, const unsigned char *helper, int size2)

// Lida com a leitura de tramas
int llread(unsigned char *packet)

// Gera o campo BCC2 para verificação de erro.
unsigned char generatebcc2(const unsigned char* data, int data_size)

// Fecha a conexão
int llclose(int showStatistics)
```

Casos de Uso Principais

As funções chamadas dependem do modo em que é executado o programa: transmissor (LITx) ou recetor (LIRx).

- llopen(): utilizada no transmissor e recetor, para inicializar a comunicação entre eles, enviando e aguardando pacotes de controlo para confirmar o início da transmissão.
- llclose(): utilizada no transmissor e recetor, para encerrar a comunicação entre o transmissor e o recetor, trocando pacotes de controlo para fechar a ligação de forma controlada e, opcionalmente, exibe estatísticas de transferência no final.

Transmissor:

- llwrite(): responsável por enviar dados na camada de ligação, construindo uma trama com a informação e os campos de controlo, aplicando byte stuffing, e gerindo o envio com retransmissões automáticas em caso de não confirmação (NACK) ou rejeição (REJ) até alcançar o número máximo de tentativas definido.
- stuff(): função chamada em llwrite() que é crucial para evitar interpretações incorretas de bytes especiais na trama de dados, garantindo a integridade da transmissão.

Recetor:

- llread(): lê e valida pacotes de dados da porta série, realizando o destuffing e verificando a integridade do pacote através de BCC2, enviando sinais de reconhecimento ou rejeição conforme necessário, e tentando novamente em caso de falha ou timeout.
- destuff(): função chamada em llread() que restaura os bytes originais que foram modificados durante o processo de stuffing, e retorna o tamanho do pacote desinflado.
- generatebcc2(): função chamada em llread() que calcula o bcc2 para garantir a integridade dos dados durante a transmissão.

Protocolo de ligação lógica

A camada de ligação de dados tem o objetivo de proporcionar comunicação entre o transmissor e o recetor, ligados por um cabo série, correspondendo à LinkLayer.
A comunicação entre os dois sistemas inicializa-se com a função llopen(). Nesta função, a porta série e o transmissor envia uma trama SET ao recetor, que responde com uma trama UA.

O envio de dados do transmissor para o recetor é feito pela função llwrite(). Para evitar conflitos com os sinais de controlo, aplica-se o *byte stuffing* -através da função stuff() - aos dados antes do envio pela porta série. Depois de enviada a trama, a função aguarda uma resposta do recetor. Esta resposta pode ser um RR (confirmação de receção) ou um REJ (pedido de reenvio em caso de erro). Caso a resposta seja um REJ ou não seja recebida, a trama é retransmitida até ser aceite ou até atingir o limite de tentativas, assegurando uma comunicação fiável entre os dispositivos.

A função llread() é responsável por receber dados do transmissor. Quando uma trama chega ao recetor, esta função processa a mensagem, aplicando *byte destuffing* - através da função destuff() - para remover bytes de escape adicionados durante o envio, garantindo que o conteúdo da trama seja interpretado corretamente. Após a receção e processamento da trama, realiza uma verificação de integridade para assegurar que os dados recebidos estão corretos. Se a trama for válida, envia uma confirmação positiva (RR) ao transmissor, indicando que a mensagem foi recebida com sucesso. Caso contrário, se for detetado um erro ou a trama não passar na verificação de integridade, responde com uma solicitação de reenvio (REJ), pedindo ao transmissor que reenvie a trama. Este mecanismo permite uma comunicação fiável, garantindo que apenas dados corretos são aceites pelo recetor.

A função llclose() é responsável por finalizar a comunicação entre o transmissor e o recetor, encerrando a ligação de forma controlada. No lado do transmissor, a função envia uma trama de término (DISC) para indicar que a transmissão de dados está concluída. O recetor, ao receber esta trama, responde com uma trama DISC, confirmando que a ligação pode ser encerrada. Após esta troca, o transmissor envia uma última trama de confirmação (UA) para finalizar a comunicação. Quando o processo de encerramento é concluído, llclose() fecha a porta série, libertando os recursos do sistema e assegurando que a

comunicação foi encerrada de forma adequada. Este procedimento garante que ambas as partes terminem a ligação de forma coordenada e evita possíveis problemas de sincronização em futuras comunicações.

Protocolo de aplicação

O protocolo de aplicação implementa a camada superior da comunicação entre o transmissor e o recetor, permitindo a transferência fiável de ficheiros através de uma ligação série. O processo começa com a inicialização da ligação, onde são recebidos parâmetros como a porta série, o papel do dispositivo (transmissor ou recetor), a taxa de baudrate, o número máximo de tentativas, o tempo limite e o nome do ficheiro a transferir. Com base nestes parâmetros, o protocolo é configurado através da estrutura LinkLayer e a ligação é estabelecida utilizando a função llopen().

No modo de transmissão (LITx), o transmissor inicia o processo verificando a existência do ficheiro e determinando o seu tamanho. Caso o ficheiro não seja encontrado ou não possa ser aberto, a ligação é encerrada. O transmissor então cria e envia um pacote de controlo de início, que contém o campo de controlo para indicar o início da transmissão, juntamente com o tamanho do ficheiro e o seu nome. Após o envio deste pacote, o transmissor começa a leitura do ficheiro e a criação dos pacotes de dados. Cada pacote é limitado pelo tamanho máximo e contém um número de sequência para garantir a ordem correta de receção. Cada pacote de dados é enviado com a função llwrite(), que aguarda confirmação antes de prosseguir com o envio do próximo pacote. Quando todos os pacotes de dados são enviados, o transmissor cria e envia um pacote de controlo de fim, indicando que a transmissão foi concluída.

No modo de receção (LIRx), o recetor aguarda e processa o pacote de controlo de início, extrai o nome e o tamanho do ficheiro e, em seguida, abre o ficheiro para gravação. A função llread() é usada para receber os pacotes de dados do transmissor. Para cada pacote recebido, o recetor verifica o número de sequência, garantindo que os pacotes sejam recebidos na ordem correta, e grava os dados no ficheiro. Quando o recetor recebe o pacote de controlo de fim, ele encerra a gravação do ficheiro, indicando que a transferência foi concluída.

Por fim, após a transmissão ou receção do ficheiro, a ligação é encerrada com a função llclose(), garantindo que todos os recursos sejam libertados de forma adequada. Este protocolo assegura uma transferência fiável dos ficheiros, utilizando pacotes de controlo para coordenar o início e o fim da transmissão, e pacotes de dados para enviar o conteúdo do ficheiro de maneira ordenada.

Validação

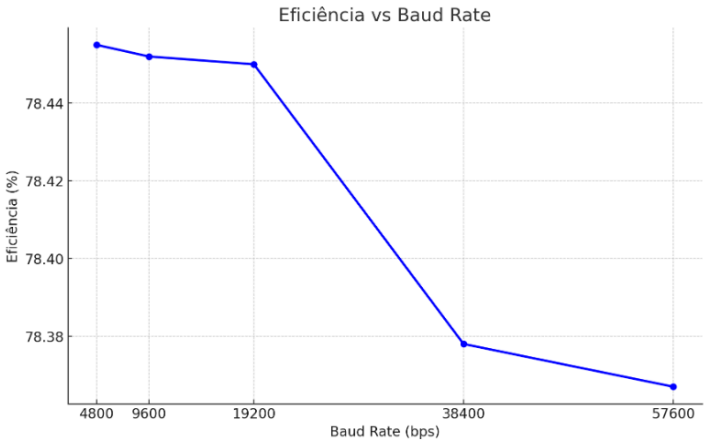
Para validar a nossa implementação do protocolo de ligação de dados, realizamos diversos testes ao longo do desenvolvimento do projeto, incluindo:

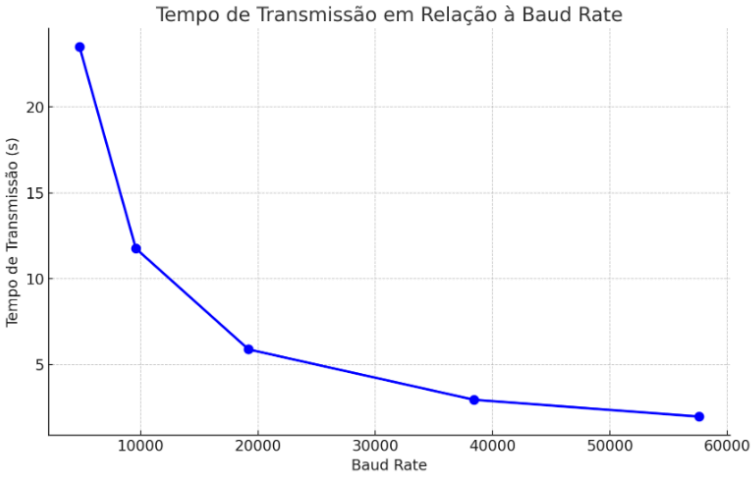
- Envio de ficheiros com diferentes tamanhos;
- Envio de ficheiros com diferentes nomes;
- Interrupção da transmissão do ficheiro (ON/OFF);
- Utilização de ruído na transmissão do ficheiro.

Todos os testes foram concluídos com sucesso, excluindo a interrupção do ficheiro, que não foi implementada corretamente.

Eficiência do Protocolo de Ligação de Dados

Com um ficheiro com tamanho fixo de 10968 bytes, a variação da baudrate demonstrou os seguintes valores:





Baudrate	Tempo(s)	Eficiência (%)	Bits recebidos por segundo
4800	23.495167	78.455	3765.838340
9600	11.748067	78.452	7531.451940
19200	5.873879	78.450	15062.816643
38400	2.939897	78.378	30122.026002
57600	1.958727	78.367	45145.726868

Concluímos assim que a eficiência diminui com o aumento da baudrate. Ou seja, o tempo total de transmissão e a velocidade são inversamente proporcionais.

Conclusões

Em suma, com a implementação da camada de aplicação (*ApplicationLayer*) e da camada de ligação de dados (*LinkLayer*), juntamente com a comunicação estabelecida entre ambas, conseguimos alcançar com sucesso o objetivo proposto.

Através da execução deste trabalho, foi possível aplicar na prática conceitos aprendidos durante as aulas, como o *byte stuffing*, e explorar a sua aplicação no contexto de comunicação de dados. Este processo permitiu não só consolidar conhecimentos técnicos, mas também aprimorar habilidades na integração de diferentes camadas de comunicação

Apêndice - Código Fonte

application_layer.c

```
#include "application_layer.h"
#include "link_layer.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void applicationLayer(const char *serialPort, const char *role, int baudRate,
                      int nTries, int timeout, const char *filename)
{

   LinkLayer test;
   strcpy(test.serialPort, serialPort);
   if (strcmp(role, "rx") == 0) {
       test.role = LlRx;
   }
   else if(strcmp(role, "tx") == 0) {
       test.role = LlTx;
   }
   else{
       return;
   }
   test.baudRate = baudRate;
   test.nRetransmissions = nTries;
   test.timeout = timeout;

   llopen(test);


   switch(test.role) {
       case (LlTx):
       {
           // VERIFICAR SE FICHEIRO EXISTE
           printf("READING FILE\n");
           FILE *file = fopen(filename, "rb");
           if (file == NULL) {
               printf("ERROR: FILE NOT FOUND\n", filename);
               llclose(0);
               return;
           }
           long int fileSize = 0;
           // VER TAMANHO DO FICHEIRO
           fseek(file, 0, SEEK_END);
           fileSize = ftell(file);
           printf("SIZE FILE: %ld bytes\n", fileSize);
           fseek(file, 0, SEEK_SET);



           //CONTROL PACKET -- start of file

           unsigned char control_start[MAX_PAYLOAD_SIZE]={0};
           int packet_size = 0;

           control_start[packet_size++] = 1; // Control field "start"
```

```
            // File size
            control_start[packet_size++] = 0; // file size T
            control_start[packet_size++] = sizeof(fileSize); // file size L
            memcpy(&control_start[packet_size], &fileSize, sizeof(fileSize)); //
file size V
            packet_size += sizeof(fileSize);  //apontar para proxima posição


            // File name
            control_start[packet_size++] = 1; // file name T
            int name_length = strlen(filename);
            control_start[packet_size++] = name_length;  // file name L
            memcpy(&control_start[packet_size], filename, name_length); // file name
V
            packet_size += name_length; //apontar para proxima posição


            printf("SENDING START CONTROL PACKET\n");
            // Enviar o pacote de controlo de início
            if (llwrite(control_start, packet_size) < 0) {
                printf("ERROR SENDING START CONTROL PACKET\n");
                fclose(file);
                llclose(1);
                return;
            }
            printf("START CONTROL PACKET SENT\n");


            // DATA PACKET

            unsigned char data_packet[MAX_PAYLOAD_SIZE + 4];
            int sequence_number = 0;
            size_t bytes_read=0;
            long bytes_remaining = fileSize;

            rewind(file);

            while (bytes_remaining > 0) {
                size_t fragment_size = (bytes_remaining > MAX_PAYLOAD_SIZE) ?
MAX_PAYLOAD_SIZE : bytes_remaining;

                bytes_read = fread(data_packet + 4, 1, fragment_size, file);
                printf("Bytes read: %zu\n", bytes_read);
                if (bytes_read<fragment_size) {
                    if (feof(file)) printf("END OF FILE\n");
                    else if (ferror(file)) perror("ERROR READING\n");
                }
                if (bytes_read <= 0) {
                    printf("ERROR READING FILE.\n");
                    fclose(file);
                    llclose(1);
                    return;
                }

                datapacket[0] = 2; // Campo C indicando "data"
                datapacket[1] = sequence_number % 100; // Número de sequência
                datapacket[2] = (bytes_read >> 8) & 0xFF; // L2 (parte alta do
tamanho)
                data_packet[3] = bytes_read & 0xFF; // L1 (parte baixa do tamanho)

                // Enviar o pacote de dados
                printf("SENDING DATA PACKET...\n");
```

```
            if (llwrite(data_packet, bytes_read + 4) < 0) {
                printf("ERROR SENDING DATA PACKET.\n");
                fclose(file);
                llclose(1);
                return;
            }
            sequence_number++;
            bytes_remaining-=bytes_read;
        }

        printf("ALL DATA PACKETS SENT.\n");


        //CONTROL PACKET -- end of file
        printf("CONTROL END PACKET ...\n");
        unsigned char control_end[256];
        memcpy(control_end, control_start, packet_size);
        control_end[0] = 3; // Campo C indicando "end"
        if (llwrite(control_end, packet_size) < 0) {
            printf("ERROR SENDING END CONTROL PACKET.\n");
            fclose(file);
            llclose(1);
            return;
        }
        printf("END CONTROL PACKET SENT.\n");



        fclose(file);
        printf("CLOSING...\n");
        llclose(1);
        break;
    }
    case (LlRx):{

        unsigned char packet[1024]={0};

        int control_packet_received = 0;
        FILE *file = NULL;
        long fileSize = 0;
        char received_filename[256] = {0};

        while (1) {
            int length = llread(packet);
            if (length < 0) {
                printf("ERROR READING PACKET.\n");
                if (file) fclose(file);
                llclose(1);
                return;
            }

            unsigned char C = packet[0]; // Campo de controle

            if (C == 1 && !control_packet_received) { // Pacote de controle
"start"
                control_packet_received = 1;

                // Processar pacote de controle de início e extrair informações
                int index = 1;
                while (index < length) {
                    unsigned char T = packet[index++];
                    unsigned char L = packet[index++];
```

```
                    if (T == 0) { // Tamanho do arquivo
                        memcpy(&fileSize, &packet[index], L);
                        index += L;
                    } else if (T == 1) { // Nome do arquivo
                        memcpy(received_filename, &packet[index], L);
                        received_filename[L] = '\0';
                        index += L;
                    }
                }

                // Abrir o arquivo para escrita
                file = fopen(filename, "wb");
                if (file == NULL) {
                    printf("ERROR OPENING FILE %s FOR WRITING.\n",
received_filename);
                    llclose(1);
                    return;
                }
                printf("START CONTROL PACKET RECEIVED: FILE %s, SIZE %ld
BYTES.\n", received_filename, fileSize);

            } else if (C == 2 && control_packet_received) { // Pacote de dados
                int sequence_number = packet[1];
                int L2 = packet[2];
                int L1 = packet[3];
                int data_size = (L2 << 8) + L1;

                // Gravar os dados no ficheiro
                fwrite(packet + 4, 1, data_size, file);
                printf("DATA PACKET RECEIVED, SEQUENCE %d, SIZE %d BYTES.\n",
sequence_number, data_size);



            } else if (C == 3 && control_packet_received) { // Pacote de
controle "end"
                printf("END CONTROL PACKET RECEIVED\n");
                break;
            }
        }

        if (file) fclose(file);

        printf("CLOSING...\n");
        llclose(1);

        break;
    }
  }
}
```

link_layer.c

```
#include "link_layer.h"
#include "serial_port.h"
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <time.h>

#define BUF_SIZE 5
#define FLAG 0x7E
#define A_T 0x03
#define A_R 0x01
#define C_SET 0X03
#define C_UA 0X07
#define C_RR0 0xAA
#define C_RR1 0xAB
#define C_REJ0 0x54
#define C_REJ1 0X55
#define C_DISC 0x0B
#define C_I0 0x00
#define C_I1 0x80
#define ESC 0x7D
#define DATA_SIZE 1024

int alarmcount = 0;
int alarmEnabled = FALSE;
volatile int STOP = FALSE;
int ret, timeout;
int fd2;
LinkLayer info;
int frame_number;
int retransmissions = 0, timeout_v ;
typedef enum {START,FLAG_RCV, A_RCV, C_RCV, BCC_OK, DATA, STOPP} States;
States s;
int rec, sended = 0;

struct timespec start, end;


void alarmHandler(int signal)
{
  alarmEnabled = FALSE;
  alarmcount++;

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

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

////////////////////////////////////////////////
// LLOPEN
////////////////////////////////////////////////
int llopen(LinkLayer connectionParameters)
{
   info=connectionParameters;
   fd2 = openSerialPort(connectionParameters.serialPort,
                        connectionParameters.baudRate);
```

```
    if (fd2 < 0 ){
        perror(connectionParameters.serialPort);
        exit(-1);
    }

    timeout = connectionParameters.timeout;
    ret = connectionParameters.nRetransmissions;

    printf("Serial port open\n") ;
    int final = 0;

    switch (connectionParameters.role) {

        case(LlTx):
        {
            s=START;
            (void) signal (SIGALRM, alarmHandler); //set alarm function handler

            unsigned char buf [BUF_SIZE] = {0};
            unsigned char receiveT[BUF_SIZE] = {0};

            clock_gettime(CLOCK_REALTIME,&start);

            buf[0] = FLAG;
            buf[1] = A_T;
            buf[2] = C_SET;
            buf[3] = (buf[1])^(buf[2]);
            buf[4] = FLAG;

            while (alarmcount < ret) {
                if (!alarmEnabled){
                    int bytesW = writeBytesSerialPort(buf, 5); //send SET in serial
port
                    printf("SET SENT: %d bytes written\n", bytesW);
                    alarm(timeout);
                    timeout_v++;
                    alarmEnabled = TRUE;
                }


                while (alarmEnabled && !STOP){
                    int bytesR_T = readByteSerialPort(receiveT); //read in serial
port
                    if (bytesR_T <= 0) continue ; //if no byte was received,
continue

                    unsigned char byteT = receiveT[0];
                    switch (s)
                    {
                        case START:
                            if (byteT==FLAG) s=FLAG_RCV;
                            break;

                        case FLAG_RCV:
                            if (byteT == FLAG) s = FLAG_RCV;
                            else if (byteT == A_R) s = A_RCV;
                            else s = START;
                            break;

                        case A_RCV:
                            if (byteT == FLAG) s = FLAG_RCV;
```

```
                        else if (byteT == C_UA) s = C_RCV;
                        else s = START;
                        break;

                    case C_RCV:
                        if (byteT == FLAG) s = FLAG_RCV;
                        else if (byteT == (A_R^C_UA)) s = BCC_OK;
                        else s = START;
                        break;

                    case BCC_OK:
                        if (byteT == FLAG) {
                            s = STOPP;
                            final=1;
                            STOP = TRUE;
                            printf("UA RECEIVED!\n");
                        } else {
                            s=START;
                        }
                        break;
                    case STOPP:
                        break;
                    default:
                        s=START;
                        break;
                }

                if (final){
                    STOP=TRUE;
                }

            }
            if (STOP) {
                break;
            }
            alarmcount++;

        }

        if (STOP) return 1;
        else return 0;
    }
    case (LlRx):
    {
        s=START;
        final=0;
        unsigned char receiveR[BUF_SIZE] = {0};
        unsigned char ua[BUF_SIZE] = {FLAG, A_R, C_UA, A_R ^ C_UA, FLAG};
        while (!STOP) {
            int bytesR_R = readByteSerialPort(receiveR);
            if (bytesR_R<=0) continue;

            unsigned char byteR = receiveR[0];
            switch (s)
            {
                case START:
                    if (byteR==FLAG) s=FLAG_RCV;
                    break;

                case FLAG_RCV:
                    if (byteR == FLAG) s = FLAG_RCV;
                    else if (byteR == A_T) s = A_RCV;
```

```
                    else s = START;
                    break;

                case A_RCV:
                    if (byteR == FLAG) s = FLAG_RCV;
                    else if (byteR == C_SET) s = C_RCV;
                    else s = START;
                    break;

                case C_RCV:
                    if (byteR == FLAG) s = FLAG_RCV;
                    else if (byteR == (A_T^C_SET)) s = BCC_OK;
                    else s = START;
                    break;

                case BCC_OK:
                    if (byteR == FLAG) {
                        s = STOPP;
                        final=1;
                        STOP = TRUE;
                        printf("SET RECEIVED!\n");
                    } else {
                        s=START;
                    }
                    break;

                default:
                    s=START;
                    break;

            }

        }
        if (STOP) {
            printf("SET RECEIVED, SENDING UA\n");
            int bytesW_R = writeBytesSerialPort(ua, BUF_SIZE);
            printf("UA SENT: %d BYTES WRITTEN\n", bytesW_R);
            return 1;
        } else return 0;
    }
  }

  return 1;
}

////////////////////////////////////////////////
// LLWRITE
////////////////////////////////////////////////

//byte stuffing
int stuff (unsigned char *stuffed, const unsigned char *helper, int size2) {
  int size = 0 ;
  stuffed[size] = helper[0];
  size++;

  for (int i = 1 ; i < size2 ; i++){
      if (helper[i] == FLAG || helper[i] == ESC){
          stuffed[size] = ESC;
          size++;
          stuffed[size] = helper[i]^0x20;
          size++;
      } else {
```

```
            stuffed[size] = helper[i];
            size++;
        }
    }
    return size;
}

int destuff(unsigned char *destuffed, const unsigned char *helper, int size2) {
    int size = 0; // Inicializa o tamanho do pacote desinflado

    // Percorre os bytes do pacote de entrada, exceto o último que é o BCC2
    for (size_t i = 0; i < size2 ; i++) {
        if (helper[i] == 0x7D) { // Byte de escape encontrado
            i++; // Avança para o próximo byte
            if (helper[i] == 0x5E) {
                destuffed[size++] = 0x7E; // Desinflação do byte original
            } else if (helper[i] == 0x5D) {
                destuffed[size++] = 0x7D; // Desinflação do byte original
            }
        } else {
            destuffed[size++] = helper[i]; // Byte não escapado
        }
    }

    // Retorna o tamanho do pacote desinflado
    return size;
}



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

    printf("\nLLWRITE() \n");
    alarmcount = 0;
    int size = 6 + bufSize;
    unsigned char frame[size];
    frame[0] = FLAG;
    frame[1] = A_T;
    frame[2] = C_I0;
    frame[3] = frame[1] ^ frame[2]; //BCC1
    int pos=4;

    unsigned char bcc2_tx = buf[0];
    for (int i = 0 ; i < bufSize; i++){
        frame[pos] = buf[i];   //adiciona data
        pos++;
        if (i > 0) bcc2_tx ^= buf[i];  // cria BCC2
    }

    frame[pos] = bcc2_tx;  //adiciona BCC2 à frame

    unsigned char stuffed[size*2];
    size = stuff(stuffed,frame,size);
    stuffed[size] = FLAG;
    size++;

    int ack = 0;
    int rej = 0 ;
    unsigned char helper = {0};
    unsigned char read = {0};
    s = START;
```

```
  (void) signal (SIGALRM, alarmHandler);

  while (alarmcount < ret){
      s = START;
      alarmEnabled = TRUE;
      alarm(timeout);
      timeout_v++;
      ack = 0;
      rej = 0;

      STOP = FALSE;
      int bytesW = writeBytesSerialPort(stuffed, size); //send information frame~
      sended += bufSize*8;
      if (bytesW==-1) {
          printf("ERROR\n");
      }
      if (bytesW==0) {
          printf("0 BYTES WRITTEN\n");
      }
      printf("Written %d bytes \n", bytesW);

      while((ack == 0 && rej == 0 && alarmEnabled) && !STOP) {
          //printf("ENTERED WHILE\n");
          int bytesR = readByteSerialPort(&read);  //receive feedback

          if (bytesR == 0) {
              continue;
          }

          switch(s) {
              case START:
                  if (read == FLAG) s = FLAG_RCV;
                  break;
              case FLAG_RCV:
                  if (read == FLAG) s = FLAG_RCV;
                  else if (read == A_T) s = A_RCV;
                  else s = START;
                  break;
              case A_RCV:
                  if (read == C_REJ0 || read == C_REJ1){
                      rej=1;
                      STOP =TRUE;
                  } else if (read == C_RR0 || read == C_RR1) {
                      ack=1;
                      alarmEnabled = FALSE;
                      s = C_RCV;
                      helper = read;
                  } else if (read == C_DISC) {

                  } else if (read == FLAG) s = FLAG_RCV;
                  else s = START;
                  break;
              case C_RCV:
                  if (read == (A_T ^ helper)) s = BCC_OK;
                  else if (read == FLAG) s = FLAG_RCV;
                  else s = START;
                  break;
              case BCC_OK:
                  if (read == FLAG) {
                      STOP = TRUE;
                  } else {
                      s = START;
```

```
                }
                break;
            default:
                s = START;
                break;
        }
        if (!helper) continue;
        else if (helper == C_RR0 || helper == C_RR1){
            ack= 1;
            rec += bufSize * 8;
        }
    }

    if (ack==1){
        return size;
    }
    if (rej==1 || alarmcount > 0) {
        alarmcount++;
        retransmissions++;
    }
  }
  return -1;
}


////////////////////////////////////////////////
// LLREAD
////////////////////////////////////////////////

unsigned char generatebcc2(const unsigned char* data, int data_size){
  unsigned char bcc2 = data[0];
  //printf("BCC2 POSIÇÃO 0: 0x%02X\n", bcc2);
  for(int i = 0 ; i < data_size-1 ; i++){

      if (i>0) bcc2 ^= data[i];
  }
  return bcc2;
}

int llread(unsigned char *packet)
{
  printf("Entered llread\n");
  alarmcount = 0;
  int packet_size = 0;
  unsigned char read[MAX_PAYLOAD_SIZE] = {0};
  unsigned char RR1[5] = {FLAG, A_T, C_RR1, A_T ^ C_RR1, FLAG};
  unsigned char RR0[5] = {FLAG, A_T, C_RR0, A_T ^ C_RR0, FLAG};
  unsigned char REJ1[5] = {FLAG, A_T, C_REJ1, A_T ^ C_REJ1, FLAG};
  unsigned char REJ0[5] = {FLAG, A_T, C_REJ0, A_T ^ C_REJ0, FLAG};
  unsigned char n;
  unsigned char bcc2_packet = 0;
  unsigned char bcc2 = 0;
  int size = 0;
  s = START;

  (void)signal(SIGALRM, alarmHandler);

  while (1) {  // Loop to retry reading packets
      alarmEnabled = TRUE;
      STOP = FALSE;
      alarm(timeout);
      size = 0;
```

```
      while (alarmEnabled && !STOP) {
          int bytesR = readByteSerialPort(read);
          if (bytesR <= 0) {
              continue; // No bytes read
          }

          unsigned char byteR = read[0];

          switch (s) {
              case START:
                  if (byteR == FLAG) s = FLAG_RCV;
                  break;
              case FLAG_RCV:
                  if (byteR == FLAG) s = FLAG_RCV;
                  else if (byteR == A_T) s = A_RCV;
                  else s = START;
                  break;
              case A_RCV:
                  if (byteR == C_I0 || byteR == C_I1) {
                      s = C_RCV;
                      frame_number = (byteR == C_I0) ? 0 : 1;
                      n = byteR;
                  } else if (byteR == FLAG) s = FLAG_RCV;
                  else s = START;
                  break;
              case C_RCV:
                  if (byteR == (A_T ^ n)) s = DATA;
                  else if (byteR == FLAG) s = FLAG_RCV;
                  else s = START;
                  break;
              case DATA:
                  if (byteR == FLAG) {  // Frame completed
                      printf("Frame detected as complete, size: %d\n", size);

                      if (size < 1) {
                          printf("Error: Incomplete packet, size: %d\n", size);
                          return -1;
                      }

                      unsigned char destuffed[MAX_PAYLOAD_SIZE + 4];
                      packet_size = destuff(destuffed, packet, size);
                      memcpy(packet, destuffed, packet_size);

                      bcc2 = packet[packet_size - 2];
                      packet_size--;
                      packet[size] = '\0';

                      bcc2_packet = generatebcc2(packet, packet_size);

                      printf("BCC2 calculated: 0x%02X, BCC2 received: 0x%02X\n",
bcc2_packet, bcc2);

                      if (bcc2 == bcc2_packet) { // Valid packet
                          printf("BCC2 matches, accepting packet, sending RR\n");
                          if (frame_number == 0) {
                              writeBytesSerialPort(RR1, 5);
                              frame_number = 1;
                              printf("RR1\n");
                          } else {
                              writeBytesSerialPort(RR0, 5);
                              frame_number = 0;
```

```
                            printf("RR0\n");
                        }
                        return packet_size; // Return valid packet size
                    } else { // Invalid packet, send REJ
                        printf("BCC2 mismatch, packet rejected, sending REJ\n");
                        if (frame_number == 0) {
                            writeBytesSerialPort(REJ0, 5);
                        } else {
                            writeBytesSerialPort(REJ1, 5);
                        }
                        break; // Exit to retry receiving a packet
                    }
                } else {
                    packet[size++] = byteR; // Collect data bytes
                    s = DATA;
                }
                break;
            default:
                break;
        }
    }

    // Handle timeout or other conditions if needed
    if (alarmcount >= ret) {
        printf("Timeout reached, retrying...\n");
        alarmcount = 0; // Reset for next attempt
    }
  }

  return -1; // In case of failure after all retries
}

////////////////////////////////////////////////
// LLCLOSE
////////////////////////////////////////////////
int llclose(int showStatistics)
{
  alarmEnabled=FALSE;
  STOP = FALSE;
  unsigned char DISC[5] = {FLAG, A_T, C_DISC, A_T^C_DISC, FLAG};
  unsigned char ua[BUF_SIZE] = {FLAG, A_R, C_UA, A_R ^ C_UA, FLAG};

  switch (info.role) {

      case(LlTx):
      {
          s=START;
          alarmcount=0;

          (void) signal (SIGALRM, alarmHandler);
          unsigned char receiveDISC[BUF_SIZE] = {0};

          while (alarmcount<ret && !STOP) {
              if (!alarmEnabled) {
                  int bytesW_DISC = writeBytesSerialPort(DISC, 5);  //SEND DISC
                  printf("Written %d bytes \n", bytesW_DISC);
                  alarm(timeout);
                  timeout_v++;
                  alarmEnabled=TRUE;
                  printf("DISC SENT first time!\n");
              }
```

```
            while (alarmEnabled==TRUE && !STOP) {
                int bytesR_DISC = readByteSerialPort(receiveDISC);
                if (bytesR_DISC <= 0) continue;
                unsigned char byteR_DISC = receiveDISC[0];

                switch (s)
                {
                    case START:
                        if (byteR_DISC==FLAG) s=FLAG_RCV;
                        break;

                    case FLAG_RCV:
                        if (byteR_DISC == FLAG) s = FLAG_RCV;
                        else if (byteR_DISC == A_T) s = A_RCV;
                        else s = START;
                        break;

                    case A_RCV:
                        if (byteR_DISC == FLAG) s = FLAG_RCV;
                        else if (byteR_DISC == C_DISC) s = C_RCV;
                        else s = START;
                        break;

                    case C_RCV:
                        if (byteR_DISC == FLAG) s = FLAG_RCV;
                        else if (byteR_DISC == (A_T^C_DISC)) s = BCC_OK;
                        else s = START;
                        break;

                    case BCC_OK:
                        if (byteR_DISC == FLAG) {
                            s = STOPP;
                            STOP = TRUE;
                            printf("DISC RECEIVED!\n");
                        } else {
                            s=START;
                        }
                        break;
                    case STOPP:
                        break;
                    default:
                        s=START;
                        break;

                }

            }
            if (STOP) {
                printf("DISC RECEIVED, SENDING UA\n");
                int bytesW_UA = writeBytesSerialPort(ua, BUF_SIZE);
                alarmEnabled = FALSE;
                printf("UA SENT: %d BYTES WRITTEN\n", bytesW_UA);
            } else {
                alarmcount++;
                if (alarmcount >= ret) return -1;
            }
        }
        break;
    }
    case (LlRx):
    {
```

```
            s=START;
            alarmcount=0;

            (void) signal (SIGALRM, alarmHandler);

            unsigned char receiveDISC[BUF_SIZE] = {0};
            unsigned char receiveUA[BUF_SIZE] = {0};

            while (alarmcount<ret && !STOP) {
                while (alarmEnabled==FALSE && !STOP) {
                    int bytesR_DISC = readByteSerialPort(receiveDISC);
                    if (bytesR_DISC<=0) continue;

                    unsigned char byteR_DISC = receiveDISC[0];

                    switch (s)
                    {
                        case START:
                            if (byteR_DISC==FLAG) s=FLAG_RCV;
                            break;

                        case FLAG_RCV:
                            if (byteR_DISC == FLAG) s = FLAG_RCV;
                            else if (byteR_DISC == A_T) s = A_RCV;
                            else s = START;
                            break;

                        case A_RCV:
                            if (byteR_DISC == FLAG) s = FLAG_RCV;
                            else if (byteR_DISC == C_DISC) s = C_RCV;
                            else s = START;
                            break;

                        case C_RCV:
                            if (byteR_DISC == FLAG) s = FLAG_RCV;
                            else if (byteR_DISC == (A_T^C_DISC)) s = BCC_OK;
                            else s = START;
                            break;

                        case BCC_OK:
                            if (byteR_DISC == FLAG) {
                                s = STOPP;
                                STOP = TRUE;
                                printf("DISC RECEIVED!\n");
                            } else {
                                s=START;
                            }
                            break;
                        case STOPP:
                            break;
                        default:
                            s=START;
                            break;

                    }
                }
                if (STOP) {
                    printf("DISC RECEIVED, SENDING DISC\n");
                    int bytesW_DISC = writeBytesSerialPort(DISC, 5);
                    printf("DISC SENT: %d BYTES WRITTEN\n", bytesW_DISC);
                    break;
                } else {
```

```
                alarmcount++;
                if (alarmcount >= ret) return -1;
                if (!alarmEnabled) alarmEnabled = TRUE;
            }
            alarmcount++;
        }
        alarmcount = 0;
        s = START;
        STOP = FALSE;

        while (alarmcount<ret && !STOP) {
            while (!alarmEnabled && !STOP) {
                int bytesR_UA = readByteSerialPort(receiveUA);
                if (bytesR_UA<=0) continue;

                unsigned char byteR_UA = receiveUA[0];

                switch (s)
                {
                    case START:
                        if (byteR_UA==FLAG) s=FLAG_RCV;
                        break;

                    case FLAG_RCV:
                        if (byteR_UA == FLAG) s = FLAG_RCV;
                        else if (byteR_UA == A_R) s = A_RCV;
                        else s = START;
                        break;

                    case A_RCV:
                        if (byteR_UA == FLAG) s = FLAG_RCV;
                        else if (byteR_UA == C_UA) s = C_RCV;
                        else s = START;
                        break;

                    case C_RCV:
                        if (byteR_UA== FLAG) s = FLAG_RCV;
                        else if (byteR_UA == (A_R^C_UA)) s = BCC_OK;
                        else s = START;
                        break;

                    case BCC_OK:
                        if (byteR_UA == FLAG) {
                            s = STOPP;
                            STOP = TRUE;
                        } else {
                            s=START;
                        }
                        break;
                    case STOPP:
                        break;

                    default:
                        s=START;
                        break;

                }

            }
            if (STOP) {
                printf("UA RECEIVED\n");
                alarm(0);
```

```
                    alarmEnabled = FALSE;
                    return 0;
                } else {
                    alarmcount++;
                    if (alarmcount >= ret) return 1;
                    if (!alarmEnabled) alarmEnabled = TRUE;
                }
            }
            break;

        }

    }

    int clstat = closeSerialPort();

    clock_gettime(CLOCK_REALTIME, &end);
    double elapsed = end.tv_sec-start.tv_sec + (end.tv_nsec-start.tv_nsec)/1e9;

    if (showStatistics) {
        printf("----Show Statistics---- \n");
        printf("Elapsed time: %f seconds \n", elapsed);
        printf("Number of retransmissions: %d\n", retransmissions);
        printf("Timeouts: %d\n", timeout_v);
        printf("Recieved bits per second %f \n", rec/elapsed);
        printf("Sended bits per second: %f \n", sended/elapsed);
        printf("Transference time: %f \n", (rec/elapsed)/(sended/elapsed));
    }

    return clstat;
}
```