

**Protocolo de Ligação de Dados**

(1º Trabalho Laboratorial)

Redes de Computadores **·** L.EIC025

2022/2023 **·** 1º Semestre

Turma: 3LEIC04

Carlos Sousa

up202005954

up202004946

up202005478

Daniela Tomás

Fábio Rocha

# Sumário

No âmbito da unidade curricular de Redes de Computadores, foi-nos proposta a realização de um protocolo de ligação de dados que consiste na transferência assíncrona de um ficheiro de um computador para outro através de um cabo série.

O principal objetivo do projeto foi concluído com sucesso, pois conseguimos implementar um protocolo de ligação de dados e testá-lo com uma aplicação que faz a transmissão de ficheiros como descrito no guião do trabalho.

# Introdução

Com este relatório, pretendemos explicar de uma forma mais detalhada e organizada o funcionamento do protocolo de ligação de dados implementado para a transmissão e receção de ficheiros através de uma porta Série RS-232. Assim, estruturámos a documentação do trabalho da seguinte forma, para que a explicação seja a mais intuitiva possível:

* **Arquitetura**  
  Demonstração dos blocos funcionais e interfaces.
* **Estrutura do código**  
  Representação das APIs, as estruturas de dados usadas, as funções e a sua interação com a arquitetura.
* **Casos de uso principais**Identificação dos principais casos de uso e sequências da chamada de funções.
* **Protocolo de ligação lógica**  
  Identificação dos principais aspetos funcionais e descrição da estratégia de implementação destes aspetos.
* **Protocolo de aplicação**  
  Identificação dos principais aspetos funcionais e descrição da estratégia de implementação destes aspetos.
* **Validação**  
  Descrição dos testes efetuados com apresentação quantificada dos resultados, se possível.
* **Eficiência do protocolo de ligação de dados**  
  Caracterização estatística da eficiência do protocolo.
* **Conclusões**  
  Síntese da informação apresentada nas secções anteriores e reflexão sobre os objetivos de aprendizagem alcançados.

# Arquitetura

O projeto está dividido em duas camadas: camada de ligação de dados e camada de aplicação. A primeira camada é responsável pela delimitação e numeração de tramas, pelo estabelecimento e terminação da ligação às portas série e pelo controlo de erros e fluxo. A camada de aplicação depende da camada de ligação de dados e é, por sua vez, responsável por enviar e receber tramas e processar e transmitir pacotes de dados ou de controlo. Como a camada de aplicação está acima da camada de ligação de dados, esta não conhece a estrutura interna do protocolo de ligação de dados, mas apenas o serviço que este oferece. Esse serviço pode ser acedido através de uma interface.

A interface do utilizador permite que o utilizador escolha a porta série, a função desempenhada pelo sistema (emissor ou recetor) e o ficheiro que pretende transferir entre os dois computadores. Esses dados interagem depois com a camada de aplicação.

# Estrutura do código

# 4.1 – Camada de ligação de dados (link\_layer.c e link\_layer.h)

Esta camada é representada por uma estrutura onde é guardada a porta série a usar na ligação, a função (emissor ou recetor), avelocidade de transmissão, o número de tentativas em caso de falha e o atraso no tempo de propagação.

typedef enum

{

    LlTx,

    LlRx,

} LinkLayerRole;

typedef struct

{

    char serialPort[50];

    LinkLayerRole role;

    int baudRate;

    int nRetransmissions;

    int timeout;

} LinkLayer;

As principais funções desta camada são a seguintes:

* ***llopen -*** Abre uma conexão usando os parâmetros da *port* definida na estrutura.
* ***llwrite -*** Envia informação num *buffer*, com um determinado tamanho, e retorna o número de caracteres escritos.
* ***llread -*** Recebe a informação num pacote e retorna o número de caracteres lidos.
* ***llclose -*** Fecha a conexão previamente aberta.

// 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);

# 4.2 – Camada de aplicação (application\_layer.c e application\_layer.h)

Esta camada é constituída por apenas uma função (***applicationLayer***) que recebe alguns dados predefinidos e outros introduzidos pelo utilizador e usa-os para transferir e processar pacotes de dados e de controlo. A camada de ligação de dados é usada nesta função para comunicar e transferir os ficheiros de acordo com os parâmetros passados.

// 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);

# Casos de uso principais

Este trabalho apresenta vários casos de uso, sendo estes a transmissão de um ficheiro entre dois computadores e a interface que permite ao emissor escolher que ficheiro enviar. Para correr o programa, o recetor deve executar o comando *make run\_rx* e o emissor *make run\_tx* dentro da pasta *code*.

No caso do emissor, a conexão entre os computadores é aberta a partir do ***llopen***, através da chamada do ***llopenT***. O pacote de controlo *start* é depois enviado e, a partir do ***llwrite***, o emissor chama a função ***header\_frame*** que, por sua vez, chama a função ***stuffing***. Em seguida, o pacote de controlo *end* é enviado e a ligação entre os computadores é fechada com o ***llclose***.

No caso do recetor, a conexão entre os computadores é aberta a partir do ***llopen***, através da chamada do ***llopenR***. O pacote de controlo *start* é recebido e, a partir do ***llread*** são recebidos os pacotes de dados. Em seguida, o pacote de controlo *end* é recebido e a ligação entre os computadores é fechada com o ***llclose***.

É importante também salientar que as funções ***state\_machine*** e ***create\_command\_frame*** são chamadas algumas vezes tanto no emissor como no recetor e têm um papel fundamental nas sequências anteriormente descritas.

# Protocolo de ligação lógica

# 6.1 - llopen

int llopen(LinkLayer connectionParameters);

O ***llopen*** tem como objetivo principal estabelecer a ligação entre o emissor e o recetor, abrindo a porta de conexão. Primeiramente, guarda as informações respetivas à porta corrente, limpa a estrutura do termios e dá setup do mesmo. Em seguida, descarta os dados escritos para o objeto referido por *fd* mas não transmitidos e dá set a uma nova porta. Finalmente, através do *LinkLayer.role* fazemos a distinção entre recetor e transmissor com as funções***llopenR*** e ***llopenT***, respetivamente***.***

No caso do ***llopenR,*** começamos por inicializar o estado a *START* para termos a certeza de que este está a começar no estado inicial e o comando *set* a *FALSE*, para verificar se o recetor se encontra pronto para receber a trama. Seguidamente, os 128 bytes do file associado ao *fd* vão sendo lidos para dentro do *buffer* a cada vez que passa no ciclo. Após isso, percorrem o buffer e a respetiva máquina de estados, verificando a cada iteração se já se encontra no estado final, no campo de endereçamento e com o comando *set* igual a *TRUE*, para assim ter a capacidade de receber a trama. Por fim, cria uma trama de comando através da chamada da função ***create\_command\_frame,*** recebendo o respetivo campo de controlo das tramas de Supervisão, que é usado para organizar qualquer tipo de trama deste tipo e deste modo serem escritos os respetivos 5 bytes com o *command value* *CVUA* para confirmar que uma conexão foi estabelecida na camada de ligação de dados e que a transmissão é de modo assíncrono.

No caso do ***llopenT***, inicializamos um inteiro *ua* a *FALSE*, de forma a controlar a resposta *UA* dentro do ciclo. Quando esta se encontra a *TRUE* significa que a resposta UA já foi recebida. Entrando no ciclo, primeiramente é controlado se o *ua* se encontra a *FALSE* e o *alarm\_count*, que conta as vezes que foi introduzido atraso na trama recebida, é menor que o número de transmissões do *LinkLayer,* sendo assim, a cada iteração é introduzido um atraso. Depois, é criada uma trama de comando através da chamada á função ***create\_command\_frame***, mas agora os argumentos passados são o *command value* de *set* (*CVSET)* responsável por indicar se é possível receber uma trama de informação e o campo de endereço (*A\_T*).

# 6.2 - llwrite

int llwrite(const unsigned char \*buf, int bufSize);

O ***llwrite*** é responsável pelo envio dos pacotes de informação, e é apenas chamada pelo transmissor. Assim, a função recebe como argumento um *buffer,* que contém a mensagem a ser enviada e o tamanho da mesma. Inicialmente, é aumentado o tamanho do buffer que vai ser usado para a escrita das tramas (*giant\_buf)*, seguidamente a trama de informação é criada, protegida por um BCC próprio e o respetivo *byte stuffing* é feito, tudo através da função ***header\_frame***. Com isto, a trama encontra-se pronta a ser enviada. De seguida, procedemos ao envio da mensagem. A resposta recebida pelo recetor pode ser de dois tipos, *CVRR* e *CVREJ*. Caso seja recebido *CVRR*, a transferência de informação ocorreu com sucesso e pode continuar a transferência, enviando a próxima trama. Caso seja recebido *CVREJ*, significa que ocorreu um erro ao ler a trama e que esta deve ser reenviada. Para que isto seja possível, à medida que as tramas são escritas o estado é alterado dentro da ***state\_machine***. No caso de o estado ser *END* (estado final), o valor de comando (*control\_value*) é comparado com os valores de *CVRR* e a *CVREJ* na função ***control\_handler***, que troca o número de sequência N(S) ou o N(R) que é o próximo número de sequência esperado na direção oposta.

# 6.3 - llread

int llread(unsigned char \*packet);

O ***llread*** é responsável por ler a trama de informação enviada pelo transmissor e realizar o *destuffing* de forma a ler a informação como era antes de realizado o *stuffing* (sem as *flags* de *escape*).

Por outro lado, enquanto o pacote ainda não foi recebido por completo, à medida que os bytes do *buffer* vão sendo lidos, a ***state\_machine*** é percorrida. Depois são também invocadas funções de introdução de erros aleatórios no cabeçalho e dados. Desta forma, a função analisa a trama recebida da seguinte forma:

* se tiver entrado no estado de rejeição, é criada uma nova trama de supervisão e enviado um *REJ*;
* se estamos na presença de um duplicado, é descartado e enviado um *RR*;
* se a trama se trata de um *DISC* significa que o envio de tramas terminou.

# 6.3 - llclose

int llclose(int showStatistics);

A função ***llclose*** é responsável por terminar a ligação através da porta série. O comportamento desta função varia caso se trate de um recetor ou transmissor. Assim, no caso do recetor, espera pela trama de supervisão *DISC* e quando a recebe, envia um *DISC* e espera por uma resposta *UA* final. No caso do transmissor, é enviada uma trama de supervisão *DISC*, recebe outro *DISC* do recetor e, por fim, envia um *UA*. A transmissão é depois fechada através da porta série.

# Protocolo de aplicação

A camada de mais alto nível é a de aplicação que é responsável pela leitura e escrita do ficheiro a enviar ou receber e pelo envio e receção dos pacotes de dados e controle.

A ***aplicationLayer*** está dividida em dois blocos, e escolhe qual deles executa, dependendo se se trata do emissor ou do recetor. Ainda antes de executar um desses blocos, é chamada a função ***llopen***, para estabelecer a conexão.

* **Recetor**

No recetor é chamada a função ***llread***, responsável por receber e ler o pacote de dados, seguidamente codifica o tamanho, o valor e o tipo do ficheiro através da função ***tlv***:

* T (um octeto) – indica qual o parâmetro (0 – tamanho do ficheiro, 1 – nome do
* ficheiro, outros valores – a definir, se necessário)
* V (número de octetos indicado em L) – valor do parâmetro
* L (um octeto) – indica o tamanho em octetos do campo V (valor do parâmetro)

Após essa função, é calculado o tamanho dos pacotes de informação e calculado o número de sequência do pacote recebido *final\_seq\_num*. A partir daí é verificada a quantidade de bytes lidos e se já chegaram ao pacote final.

* **Emissor**

No emissor, para garantir que os dados que foram recebidos apresentam coesão, são feitas algumas verificações. Em primeiro lugar, os atributos do file são colocados no buffer, depois o emissor preparar as tramas de início (*packet[0]* = *START*) e as de fim (*packet[0]* = *END*). Posteriormente, é adicionado o cabeçalho ao *packet*, no qual fica a informação relativa ao tamanho de dados que o pacote transporta. A utilidade deste cabeçalho é provada pelo facto de este permitir verificação caso o *file* tenha sido corrompido.

# Validação

* **Testes efetuados**

1. Envio de um ficheiro;
2. Envio de um ficheiro com pacotes de tamanhos diferentes;
3. Envio de um ficheiro com diferentes valores de *baudrate* (capacidade de ligação);
4. Envio de um ficheiro com *timeouts* diferentes;
5. Envio de um ficheiro com número de retransmissões diferentes;
6. Envio de ficheiros de tamanhos diferentes;
7. Interromper a ligação da porta série enquanto envia o ficheiro;
8. Introduzir ruído na porta série enquanto envia o ficheiro.

* **Resultados obtidos**

Todos os testes foram concluídos com sucesso, exceto os testes 6, 7 e 8 que falharam provavelmente devido a problemas de alocação de memória.

# Eficiência do protocolo de ligação de dados

Nota: Todas as tabelas e gráficos encontram-se em Anexo II – Eficiência.

# - Variação do tamanho dos pacotes

A eficiência aumenta com o aumento do tamanho dos pacotes, porque são necessários menos pacotes e, por isso, menos dados (e.g., cabeçalhos) são enviados.

# - Variação do *baudrate*

Quanto maior a capacidade de ligação (*baudrate*) menor é a eficiência do protocolo.

# - Variação do *timeout*

Quanto maior o tempo de propagação menor é a eficiência, porque o aumento do tempo faz com que as tramas demorem mais a ser enviadas e recebidas.

# Conclusões

Podemos concluir que tanto o objetivo principal, que consiste no envio de um ficheiro entre dois computadores através do uso da porta série RS-232, como os restantes objetivos referentes à independência total de camadas foram concluídos.

Este trabalho contribuiu principalmente para uma melhor compreensão acerca da independência de camadas em sistemas e a forma como estes estão organizados. A camada de ligação de dados, para além de oferecer um serviço de comunicação fiável entre dois sistemas ligados por um canal de transmissão, transmite a informação sem que ocorra o processamento do respetivo conteúdo através de um mecanismo de controlo de fluxo e de erros. Por outro lado, a camada de aplicação é responsável pela transferência de um ficheiro usando o serviço fiável oferecido pelo protocolo de ligação de dados para envio de pacotes.

Em suma, baseando-nos nas características do nosso sistema, podemos afirmar com toda a convicção que adquirimos conhecimentos de extrema relevância ao longo da sua implementação. Quanto á funcionalidade dos nossos protocolos, temos confiança na respetiva aptidão do sistema.

# Anexo I - Código Fonte

# 11.1 – Makefile

# Makefile to build the project

# NOTE: This file must not be changed.

# Parameters

CC = gcc

CFLAGS = -Wall

SRC = src/

INCLUDE = include/

BIN = bin/

CABLE\_DIR = cable/

TX\_SERIAL\_PORT = /dev/ttyS10

RX\_SERIAL\_PORT = /dev/ttyS11

TX\_FILE = penguin.gif

RX\_FILE = penguin-received.gif

# Targets

.PHONY: all

all: $(BIN)/main $(BIN)/cable

$(BIN)/main: main.c $(SRC)/\*.c

    $(CC) $(CFLAGS) -o $@ $^ -I$(INCLUDE)

$(BIN)/cable: $(CABLE\_DIR)/cable.c

    $(CC) $(CFLAGS) -o $@ $^

.PHONY: run\_tx

run\_tx: $(BIN)/main

    ./$(BIN)/main $(TX\_SERIAL\_PORT) tx $(TX\_FILE)

.PHONY: run\_rx

run\_rx: $(BIN)/main

    ./$(BIN)/main $(RX\_SERIAL\_PORT) rx $(RX\_FILE)

.PHONY: run\_cable

run\_cable: $(BIN)/cable

    ./$(BIN)/cable

.PHONY: check\_files

check\_files:

    diff -s $(TX\_FILE) $(RX\_FILE) || exit 0

.PHONY: clean

clean:

    rm -f $(BIN)/main

    rm -f $(BIN)/cable

    rm -f $(RX\_FILE)

# 11.2 – main.c

// Main file of the serial port project.

// NOTE: This file must not be changed.

#include <stdio.h>

#include <stdlib.h>

#include "application\_layer.h"

#define BAUDRATE 9600

#define N\_TRIES 3

#define TIMEOUT 4

// Arguments:

//   $1: /dev/ttySxx

//   $2: tx | rx

//   $3: filename

int main(int argc, char \*argv[])

{

    if (argc < 4)

    {

        printf("Usage: %s /dev/ttySxx tx|rx filename\n", argv[0]);

        exit(1);

    }

    const char \*serialPort = argv[1];

    const char \*role = argv[2];

    const char \*filename = argv[3];

    printf("Starting link-layer protocol application\n"

           "  - Serial port: %s\n"

           "  - Role: %s\n"

           "  - Baudrate: %d\n"

           "  - Number of tries: %d\n"

           "  - Timeout: %d\n"

           "  - Filename: %s\n",

           serialPort,

           role,

           BAUDRATE,

           N\_TRIES,

           TIMEOUT,

           filename);

    applicationLayer(serialPort, role, BAUDRATE, N\_TRIES, TIMEOUT, filename);

    return 0;

}

# 11.3 – application\_layer.c

// Application layer protocol implementation

#include <unistd.h>

#include <string.h>

#include <stdio.h>

#include "application\_layer.h"

#include "link\_layer.h"

#include "utils.h"

#include "alarm.h"

#include <stdlib.h>

#include <sys/stat.h>

unsigned char packet[PCK\_SIZE+30];

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

    LinkLayer connectionParameters;

    strcpy(connectionParameters.serialPort, serialPort);

    if(strcmp(role,"rx") == 0) {

        connectionParameters.role = LlRx;

    }

    else if(strcmp(role,"tx") == 0) {

        connectionParameters.role = LlTx;

    }

    else {

        perror(role);

        exit(-1);

    }

    connectionParameters.baudRate = baudRate;

    connectionParameters.nRetransmissions = nTries;

    connectionParameters.timeout = timeout;

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

    if(llopen(connectionParameters) == -1) {

      perror("Error opening a connection using the port parameters");

      exit(-1);

    }

    if(connectionParameters.role == LlRx) {

        int file\_size = 0, size\_rx = 0;

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

        int bytes = llread(packet);

        int type,length,\*value;

        if(packet[0] == CTRLSTART) {

            int tlv\_size = 1;

            while(tlv\_size < bytes) {

                tlv\_size += tlv(packet + tlv\_size, &type, &length, &value);

                if(type == 0){

                    file\_size =\* value;

                    //printf("File size: %d\n",file\_size);

                }

            }

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

            if(file == NULL) {

                perror("Cannot open the file\n");

                exit(-1);

            }

            else {

                printf("Control packet received\n");

            }

            int final\_seq\_num = 0;

            while(size\_rx < file\_size) {

                int bytes;

                if((bytes = llread(packet)) == -1){

                    perror("llread\n");

                    exit(-1);

                }

                if(packet[0] == CTRLEND){

                    perror("CTRL\_END failed\n");

                    exit(-1);

                }

                if(packet[0] == CTRLDATA){

                    if(bytes < 5) {

                        perror("CTRLDATA failed\n");

                        exit(-1);

                    }

                    else if(packet[1] != final\_seq\_num){

                        perror("final\_seq\_num failed\n");

                        exit(-1);

                    }

                    else{

                        unsigned long size = packet[3] + packet[2]\*256;

                        if(bytes-4 != size) {

                            perror("Header deprecated\n");

                            exit(-1);

                        }

                        fwrite(packet + 4, 1, size, file);

                        size\_rx += size;

                        printf("Packet %d received\n", final\_seq\_num);

                        final\_seq\_num++;

                    }

                }

            }

            int bytes\_read = llread(packet);

            if(bytes\_read == -1) {

                perror("llread\n");

                exit(-1);

            }

            else if(bytes\_read < 1) {

                perror("Short packet\n");

                exit(-1);

            }

            if(packet[0] != CTRLEND){

                printf("CTRLEND failed\n");

            }

            else{

                printf("Received end packet\n");

                printf("Transmission ending...\n");

            }

            fclose(file);

        }

        else {

            perror("Start packet failed\n");

            exit(-1);

        }

    }

    else if(connectionParameters.role == LlTx) {

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

        if(file == NULL) {

            perror("Error openning the file\n");

            exit(-1);

        }

        struct stat st;

        int file\_size = (stat(filename, &st) == 0) ? st.st\_size : 0;

        packet[0] = CTRLSTART;

        packet[1] = 0;

        packet[2] = sizeof(long);

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

        \*((long\*)(packet + 3)) = file\_size;

        if(llwrite(packet,10) == -1){

            perror("llwrite\n");

            exit(-1);

        }

        int bytes\_tx = 0;

        unsigned char i = 0;

        do {

            unsigned long total\_bytes;

            if(file\_size-bytes\_tx < PCK\_SIZE) {

                total\_bytes = fread(packet + 4, 1, file\_size-bytes\_tx, file);

            }

            else {

                total\_bytes = fread(packet + 4, 1, PCK\_SIZE, file);

            }

            packet[0] = CTRLDATA;

            packet[1] = i;

            packet[2] = total\_bytes >> 8;

            packet[3] = total\_bytes % 256;

            if(llwrite(packet, total\_bytes + 4) == -1){

                perror("llwrite\n");

                exit(-1);

                break;

            }

            printf("Packet %i sent\n",i);

            bytes\_tx += total\_bytes;

            i++;

        }while(bytes\_tx < file\_size);

        packet[0] = CTRLEND;

        if(llwrite(packet,1) == -1){

            perror("llwrite\n");

            exit(-1);

        }

        fclose(file);

    }

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

    if(llclose(TRUE) == -1) {

        perror("llclose");

        exit(-1);

    }

}

# 11.4 – 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\_

# 11.5 – link\_layer.c

// Link layer protocol implementation

#include <termios.h>

#include <stdio.h>

#include <unistd.h>

#include <fcntl.h>

#include <signal.h>

#include <string.h>

#include <stdlib.h>

#include "link\_layer.h"

#include "state\_machine.h"

#include "alarm.h"

#include "utils.h"

// MISC

#define \_POSIX\_SOURCE 1 // POSIX compliant source

struct termios oldtio;

struct termios newtio;

unsigned char buffer[MAX\_PCK\_SIZE], \*giant\_buf = NULL;

int giant\_buf\_size = 0;

unsigned char sflag = FALSE;

int fd;

int disc = FALSE;

LinkLayer cp;

////////////////////////////////////////////////

// LLOPEN

////////////////////////////////////////////////

int llopen(LinkLayer connectionParameters) {

    cp = connectionParameters;

    // Open serial port device for reading and writing and not as control\_valueling tty

    // because we don't want to get killed if linenoise sends CTRL-C.

    fd = open(cp.serialPort, O\_RDWR | O\_NOCTTY);

    if (fd < 0) {

        perror(cp.serialPort);

        exit(-1);

    }

    // Save current port settings

    if (tcgetattr(fd, &oldtio) == -1) {

        perror("Error getting new port settings.\n");

        exit(-1);

    }

    // Clear struct for new port settings

    bzero(&newtio, sizeof(newtio));

    newtio.c\_cflag = cp.baudRate | CS8 | CLOCAL | CREAD;

    newtio.c\_iflag = IGNPAR;

    newtio.c\_oflag = 0;

    newtio.c\_lflag = 0;

    newtio.c\_cc[VTIME] = 0; // Inter-character timer unused

    newtio.c\_cc[VMIN] = 1;  // Blocking read until 1 char received

    if(tcflush(fd, TCIOFLUSH) == -1) {

        perror("Error flushing the data.\n");

        exit(-1);

    }

    // Set new port settings

    if (tcsetattr(fd, TCSANOW, &newtio) == -1) {

        perror("Error setting new port settings.\n");

        exit(-1);

    }

    signal(SIGALRM,alarmHandler);

    switch(cp.role) {

         case LlRx:

            return llopenR(fd);

            break;

         case LlTx:

            return llopenT(fd,cp.nRetransmissions,cp.timeout);

            break;

         default:

            break;

    }

    return -1;

}

////////////////////////////////////////////////

// LLWRITE

////////////////////////////////////////////////

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

    if(giant\_buf\_size < bufSize\*2 + 10) {

        giant\_buf = (giant\_buf\_size == 0) ? malloc(bufSize\*2 + 10) : realloc(giant\_buf, bufSize\*2+10);

    }

    int fsize = header\_frame(giant\_buf, buf, bufSize, A\_T, control\_handler(CVDATA, sflag));

    int bytes\_written = write\_cycle(fsize);

    int isReceivedP = FALSE;

    int nRetransmissions = 0;

    data = NULL;

    alarm\_enabled = TRUE;

    alarm(cp.timeout);

    while(isReceivedP == FALSE){

        if(!alarm\_enabled) {

            alarm\_enabled = TRUE;

            alarm(cp.timeout);

            if(nRetransmissions > 0) {

                printf("Time-out\n");

                return -1;

            }

            else if(cp.nRetransmissions == nRetransmissions){

                printf("Number of retransmissions allowed exceeded\n");

                return -1;

            }

            bytes\_written += write\_cycle(fsize);

            nRetransmissions++;

        }

        int bytes\_read;

        if((bytes\_read = read(fd, buffer, MAX\_PCK\_SIZE)) == -1) {

            printf("Couldn't read (llwrite)\n");

            return -1;

        }

        int j = 0;

        do{

            state\_machine(buffer[j]);

            if(fstate == END){

                if(address == A\_T && (control\_value == control\_handler(CVRR, FALSE) || control\_value == control\_handler(CVRR, TRUE))){

                    isReceivedP = TRUE;

                    if(control\_value == control\_handler(CVRR, sflag)) {

                        printf("Requesting next packet...\n");

                    }

                }

                if(address == A\_T && control\_value == control\_handler(CVREJ, sflag)) {

                    printf("Requesting retransmission...\n");

                    nRetransmissions = 0;

                }

            }

            j++;

        }while(j < bytes\_read && isReceivedP == FALSE && alarm\_enabled);

    }

    if(sflag) {

        sflag = FALSE;

    }

    else {

        sflag = TRUE;

    }

    //printf("%d chars written\n",bytes\_written);

    return bytes\_written;

}

////////////////////////////////////////////////

// LLREAD

////////////////////////////////////////////////

int llread(unsigned char \*packet) {

    if(giant\_buf\_size < MAX\_PCK\_SIZE){

        giant\_buf = (giant\_buf\_size == 0)? malloc(MAX\_PCK\_SIZE): realloc(giant\_buf, MAX\_PCK\_SIZE);

    }

    int isReceivedP = FALSE;

    data = packet;

    int bytes\_read;

    while(isReceivedP == FALSE){

        if((bytes\_read = read(fd, giant\_buf, MAX\_PCK\_SIZE)) == -1) {

            printf("Couldn't read (llread)\n");

            return -1;

        }

        int i = 0;

        do {

            state\_machine(giant\_buf[i]);

            if(address == A\_T) {

                if(fstate == REJECT){

                    create\_command\_frame(buffer,((control\_value == control\_handler(CVDATA, FALSE)) ? control\_handler(CVREJ, FALSE):control\_handler(CVREJ,TRUE)),A\_T);

                    write(fd, buffer, 5);

                    printf("REJ sent\n");

                }

                else if(fstate == END) {

                    if(control\_value == CVSET){

                        create\_command\_frame(buffer, CVUA, A\_T);

                        write(fd, buffer, 5);

                        printf("UA sent\n");

                    }

                    else if(control\_value == control\_handler(CVDATA, sflag)){

                        sflag = (sflag)? FALSE : TRUE;

                        create\_command\_frame(buffer, control\_handler(CVRR, sflag), A\_T);

                        write(fd, buffer, 5);

                        printf("RR %i sent\n", sflag);

                        return size;

                    }

                    else {

                        create\_command\_frame(buffer,control\_handler(CVRR, sflag), A\_T);

                        write(fd, buffer, 5);

                        printf("RR %i sent requesting retransmission\n",sflag);

                    }

                }

            }

            if(control\_value == CVDISC) {

                disc = TRUE;

                create\_command\_frame(buffer, (control\_value == control\_handler(CVDATA, FALSE) ? control\_handler(CVREJ, FALSE):control\_handler(CVREJ, TRUE)), A\_T);

                write(fd, buffer, 5);

                printf("DISC received\n");

                return -1;

                break;

            }

            i++;

        }while(i < bytes\_read && !isReceivedP);

    }

    return -1;

}

////////////////////////////////////////////////

// LLCLOSE

////////////////////////////////////////////////

int llclose(int showStatistics) {

    signal(SIGALRM, alarmHandler);

    if(giant\_buf\_size > 0) {

        free(giant\_buf);

    }

    if(cp.role == LlRx) {

        int bytes\_read;

        while(disc == FALSE){

            if((bytes\_read = read(fd, buffer, MAX\_PCK\_SIZE)) == -1) {

                printf("Couldn't read (llclose)\n");

                return -1;

            }

            int i = 0;

            do{

                state\_machine(buffer[i]);

                if(fstate == END && address == A\_T && control\_value == CVDISC) {

                    disc = TRUE;

                }

                i++;

            }while(i < bytes\_read && disc == FALSE);

        }

        if(disc == TRUE) {

            printf("DISC received\n");

        }

        create\_command\_frame(buffer, CVDISC, A\_T);

        if(write(fd, buffer, 5) == -1) {

            printf("Couldn't write (llclose)\n");

            return -1;

        }

        printf("DISC sent\n");

        int ua = FALSE;

        while(ua == FALSE) {

            if((bytes\_read = read(fd, buffer, MAX\_PCK\_SIZE)) == -1) {

                printf("Couldn't read (llclose)");

                return -1;

            }

            int i = 0;

            do{

                state\_machine(buffer[i]);

                if(fstate == END && address == A\_T && control\_value == CVUA) {

                    ua = TRUE;

                }

                i++;

            }while(i < bytes\_read && ua == FALSE);

        }

        if(ua == TRUE) {

            printf("UA received\n");

        }

    }

    else if(cp.role == LlTx) {

        int bytes\_read, isDisc = FALSE;

        alarm\_count = 0;

        while(alarm\_count < cp.nRetransmissions && isDisc == FALSE){

            alarm(cp.timeout);

            alarm\_enabled = TRUE;

            if(alarm\_count > 0) {

                printf("Time-out.\n");

                return -1;

            }

            create\_command\_frame(buffer, CVDISC, A\_T);

            printf("DISC sent\n");

            if(write(fd, buffer, 5) == -1) {

                printf("Couldn't write (llclose)\n");

                return -1;

            }

            while(alarm\_enabled && isDisc == FALSE){

                if((bytes\_read = read(fd, buffer, MAX\_PCK\_SIZE)) == -1) {

                    printf("Couldn't read (llclose)\n");

                    return -1;

                }

                int i = 0;

                do {

                    state\_machine(buffer[i]);

                    if(fstate == END && address == A\_T && control\_value == CVDISC) {

                        isDisc = TRUE;

                    }

                    if(fstate == END && address == A\_T && (control\_value == control\_handler(CVRR, FALSE) || control\_value == control\_handler(CVRR, TRUE) || control\_value == control\_handler(CVREJ,FALSE) || control\_value == control\_handler(CVREJ,TRUE))){

                        alarm\_count = 0;

                    }

                    i++;

                }while(i < bytes\_read && isDisc == FALSE);

            }

        }

        if(isDisc == TRUE) {

            printf("DISC received\n");

            create\_command\_frame(buffer, CVUA, A\_T);

            write(fd, buffer, 5);

            printf("UA sent\n");

            sleep(1);

        }

    }

    if (tcsetattr(fd, TCSANOW, &oldtio) == -1) {

        printf("tcsetattr\n");

        return -1;

    }

    close(fd);

    return 1;

}

# 11.6 – link\_layer.h

// Link layer header.

// NOTE: This file must not be changed.

#ifndef \_LINK\_LAYER\_H\_

#define \_LINK\_LAYER\_H\_

typedef enum

{

    LlTx,

    LlRx,

} LinkLayerRole;

typedef struct

{

    char serialPort[50];

    LinkLayerRole role;

    int baudRate;

    int nRetransmissions;

    int timeout;

} LinkLayer;

extern unsigned char buffer[128], \*giant\_buf;

extern int giant\_buf\_size;

typedef enum {CVRR, CVREJ, CVDATA} ControlV;

extern ControlV cv;

// 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\_

# 11.7 – state\_machine.c

#include <stdio.h>

#include <string.h>

#include "state\_machine.h"

#include "link\_layer.h"

#include "utils.h"

unsigned char address;

unsigned char control\_value;

unsigned char bcc;

unsigned char \*data;

unsigned int size;

FState fstate;

void state\_machine(unsigned char byte){

    switch (fstate){

        case REJECT:

        case END:

            fstate = START;

        case START:

            //printf("START\n");

            if(byte == F)  {

                fstate = FLAG;

            }

            break;

        case FLAG:

            //printf("FLAG\n");

            size = 0;

            if(byte != F) {

                if(byte == A\_T || byte == A\_R){

                    fstate = ADDRESS;

                    address = byte;

                }

                else {

                    fstate = START;

                }

            }

            break;

        case ADDRESS:

            //printf("ADDRESS\n");

            if(byte == F) {

                fstate = FLAG;

            }

            else if( byte == control\_handler(CVREJ,FALSE) || byte == control\_handler(CVRR,FALSE) || byte == control\_handler(CVREJ,TRUE) || byte == control\_handler(CVRR,TRUE)

            || byte == control\_handler(CVDATA,FALSE) || byte == control\_handler(CVDATA,TRUE) || byte == CVDISC || byte == CVSET || byte == CVUA){

                control\_value = byte;

                bcc = address ^ control\_value;

                fstate = CONTROL;

            }

            else {

                fstate = START;

            }

            break;

        case CONTROL:

            //printf("CONTROL\n");

            if(byte == F) {

                fstate = FLAG;

            }

            else if(byte == bcc) {

                fstate = BCC1\_OK;

            }

            else {

                fstate = START;

            }

            break;

        case BCC1\_OK:

            //printf("BCC1\_OK\n");

            if(byte == F){

                if(control\_value == control\_handler(CVDATA, FALSE) || control\_value == control\_handler(CVDATA, TRUE)) {

                    fstate = F;

                    break;

                }

                fstate = END;

            }

            else if(((control\_value == control\_handler(CVDATA, FALSE) || control\_value == control\_handler(CVDATA, TRUE)) && data != NULL)){

                size = 0;

                if(byte == ESC) {

                    bcc = 0;

                    fstate = ESCAPE;

                    break;

                }

                data[size] = byte;

                size++;

                bcc = byte;

                fstate = SUCCESS;

            }

            else {

             fstate = START;

            }

            break;

        case BCC2\_OK:

            //printf("BCC2\_OK\n");

            if(byte == F) {

                fstate = END;

            }

            else if(byte == 0) {

                data[size] = bcc;

                size++;

                bcc = 0;

                if(byte == ESC) {

                    data[size] = bcc;

                    size++;

                    bcc = 0;

                    fstate = ESCAPE;

                }

            }

            else {

                data[size] = bcc;

                size++;

                data[size] = byte;

                size++;

                bcc = byte;

                fstate = SUCCESS;

            }

            break;

        case SUCCESS:

            //printf("SUCCESS\n");

            if(byte == F) {

                fstate = REJECT;

            }

            else if(byte == ESC) {

                fstate = ESCAPE;

            }

            else if(byte == bcc) {

                fstate = BCC2\_OK;

            }

            else {

                data[size] = byte;

                size++;

                bcc ^= byte;

            }

            break;

        case ESCAPE:

            //printf("ESCAPE\n");

            if(byte == F) {

                fstate = REJECT;

            }

            else if(byte == ESC\_F) {

                if(bcc == F){

                    fstate = BCC2\_OK;

                    break;

                }

                bcc ^= F;

                data[size] = F;

                size++;

                fstate = SUCCESS;

            }

            else if(byte == ESC\_E) {

                if(bcc == ESC) {

                    fstate = BCC2\_OK;

                    break;

                }

                bcc ^= ESC;

                data[size] = ESC;

                size++;

                fstate = SUCCESS;

            }

            else {

                fstate = START;

            }

            break;

    }

}

# 11.8 – state\_machine.h

// Statemachine header.

#ifndef \_STATEMACHINE\_H\_

#define \_STATEMACHINE\_H\_

extern unsigned char address, control\_value, bcc, \*data;

extern unsigned int size;

typedef enum {START, FLAG, ADDRESS, CONTROL, BCC1\_OK, BCC2\_OK, SUCCESS, ESCAPE, END, REJECT} FState;

extern FState fstate;

void state\_machine(unsigned char byte);

#endif // \_STATEMACHINE\_H\_

# 11. 9 – utils.c

#include "utils.h"

#include "state\_machine.h"

#include "link\_layer.h"

#include "alarm.h"

#include <stdlib.h>

#include <stdio.h>

#include <unistd.h>

int llopenR(int fd) {

    fstate = START;

    alarm\_count = 0;

    int set = FALSE, byte;

    while(set == FALSE) {

        if((byte = read(fd, buffer, MAX\_PCK\_SIZE)) == -1) {

            perror("Couldn't read (llopen)\n");

            exit(-1);

        }

        int i = 0;

        do {

            state\_machine(buffer[i]);

            if(fstate == END && address == A\_T) {

                if(control\_value == CVSET) {

                    set = TRUE;

                }

                else if(control\_value == CVDISC) {

                    disc = TRUE;

                    printf("DISC Received\n");

                    return -1;

                }

            }

            i++;

        }while(i < byte && set == FALSE);

    }

    if(set == TRUE) {

        printf("Set Received\n");

    }

    create\_command\_frame(buffer, CVUA, A\_T);

    write(fd, buffer, 5);

    printf("UA Sent\n");

    return 1;

}

int llopenT(int fd, int nRetransmissions, int timeout) {

        fstate = START;

        alarm\_count = 0;

        int ua = FALSE, byte;

        while(ua == FALSE && alarm\_count < nRetransmissions) {

            alarm(timeout);

            alarm\_enabled = TRUE;

            if(alarm\_count > 0) {

                printf("Time-out\n");

            }

            create\_command\_frame(buffer, CVSET, A\_T);

            printf("SET sent\n");

            write(fd, buffer, 5);

            while(alarm\_enabled && ua == FALSE){

                if((byte = read(fd, buffer, MAX\_PCK\_SIZE)) == -1) {

                    printf("Couldn't read (llopen)\n");

                    return -1;

                }

                int i = 0;

                do {

                    state\_machine(buffer[i]);

                    if(fstate == END && address == A\_T && control\_value == CVUA) {

                        ua = TRUE;

                    }

                    i++;

                }while(i < byte && ua == FALSE);

            }

        }

        if(ua == TRUE) {

            printf("UA Received\n");

            return 1;

        }

        return -1;

}

int stuffing(const unsigned char \*buffer, int bufSize, unsigned char\* oct, unsigned char \*bcc){

    int size = 0, i = 0;

    do{

        if(bcc != NULL) {

            \*bcc ^= buffer[i];

        }

        if(buffer[i] == ESC) {

            oct[size++] = ESC;

            oct[size++] = ESC\_E;

            return size;

        }

        else if(buffer[i] == F) {

            oct[size++] = ESC;

            oct[size++] = ESC\_F;

            return size;

        }

        oct[size++] = buffer[i];

        i++;

    }while(i < bufSize);

    return size;

}

void create\_command\_frame(unsigned char\* buf, unsigned char control\_value, unsigned char address){

    buf[0] = F;

    buf[1] = address;

    buf[2] = control\_value;

    buf[3] = address ^ control\_value;

    buf[4] = F;

}

int header\_frame(unsigned char\* buf, const unsigned char\* data,unsigned int data\_size, unsigned char address, unsigned char control\_value){

    buf[0] = F;

    buf[1] = address;

    buf[2] = control\_value;

    buf[3] = address ^ control\_value;

    int new\_size = 0, i = 0;

    unsigned char bcc = 0;

    do{

        new\_size += stuffing(data + i, 1, buf + new\_size + HEADER\_SIZE, &bcc);

        i++;

    }while(i < data\_size);

    new\_size += stuffing(&bcc,1, buf + new\_size + HEADER\_SIZE, NULL);

    buf[new\_size + HEADER\_SIZE] = F;

    new\_size += 5;

    return new\_size;

}

int tlv(unsigned char \*address, int\* type, int\* length, int\*\* value){

    \*type = address[0];

    \*length = address[1];

    \*value = (int\*)(address + 2);

    return 2 + \*length;

}

int write\_cycle(int size) {

    int i = 0, wr;

    while(i < size) {

        if((wr = write(fd, giant\_buf + i, size - i)) == -1) {

            perror("Couldn't write (llwrite).\n");

            exit(-1);

        }

        i += wr;

    }

    return i;

}

int control\_handler(ControlV cv, int R\_S) {

    switch (cv) {

        case CVRR:

            if(R\_S % 2 != 0) {

                return 0b10000101;

            }

            else {

                return 0b00000101;

            }

            break;

        case CVREJ:

            if(R\_S % 2 != 0){

                return 0b10000001;

            }

            else {

                return 0b00000001;

            }

            break;

        case CVDATA:

            if(R\_S % 2 != 0) {

                return 0b01000000;

            }

            else {

                return 0b00000000;

            }

            break;

        default:

            break;

    }

    return -1;

}

# 11.10 – utils.h

#ifndef \_UTILS\_H\_

#define \_UTILS\_H\_

#include "link\_layer.h"

#define F 0x7e

#define ESC 0x7d

#define ESC\_F 0x5e

#define ESC\_E 0x5d

#define A\_T 0x03

#define A\_R 0x01

#define HEADER\_SIZE 4

#define PCK\_SIZE 512

#define MAX\_PCK\_SIZE 128

#define CTRLDATA 1

#define CTRLSTART 2

#define CTRLEND 3

#define CVSET 0x03

#define CVDISC 0x0b

#define CVUA 0x07

extern int alarm\_count;

extern int alarm\_enabled;

extern int disc;

extern unsigned char buffer[MAX\_PCK\_SIZE];

extern int fd;

int llopenR(int fd);

int llopenT(int fd, int nRetransmissions, int timeout);

int stuffing(const unsigned char \*buf, int bufSize, unsigned char\* dest, unsigned char \*bcc);

void create\_command\_frame(unsigned char\* buf, unsigned char control\_value, unsigned char address);

int header\_frame(unsigned char\* framebuf, const unsigned char\* data, unsigned int data\_size, unsigned char address, unsigned char control\_value);

int tlv(unsigned char \*address, int\* type, int\* length, int\*\* value);

int write\_cycle(int size);

int control\_handler(ControlV cv, int R\_S);

#endif // \_UTILS\_H

# 11.11 – alarm.c

#include <unistd.h>

#include <signal.h>

#include <stdio.h>

#include "link\_layer.h"

#include "alarm.h"

int alarm\_enabled = FALSE;

int alarm\_count = 0;

void alarmHandler(int signal) {

    alarm\_enabled = FALSE;

    alarm\_count++;

    printf("Alarm Counter: %d\n", alarm\_count);

}

int setAlarm(int timeout) {

    (void)signal(SIGALRM, alarmHandler);

    while (alarm\_count <= timeout) {

        if (alarm\_enabled == FALSE) {

            alarm(timeout);

            alarm\_enabled = TRUE;

        }

    }

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

    printf("Operation ending\n");

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

    return 0;

}

# 11.12 – alarm.h

// Alarm header.

#ifndef \_ALARM\_H\_

#define \_ALARM\_H\_

void alarmHandler(int signal);

int setAlarm(int timeout);

extern int alarm\_count;

extern int alarm\_enabled;

#endif // \_ALARM\_H\_

# Anexo II – Eficiência

# – Variação do tamanho dos pacotes

Chart, line chart

Description automatically generated

C = 9600 b/s

|  |  |  |  |
| --- | --- | --- | --- |
| **Tamanho de cada pacote(bytes)** | **Tempo (s)** | **R (bits/tempo)** | **S (R/C)** |
| 32 | 20,19823 | 4344,143026 | 0,452514899 |
| 64 | 16,56327 | 5297,504659 | 0,551823402 |
| 128 | 13,42891 | 6533,96292 | 0,680621138 |
| 256 | 12,92356 | 6789,460489 | 0,707235468 |
| 512 | 12,53882 | 6997,787671 | 0,728936216 |
| 1024 | 11,87824 | 7386,952949 | 0,769474266 |
| 2048 | 11,75369 | 7465,230068 | 0,777628132 |
| 4096 | 11,72253 | 7485,073615 | 0,779695168 |

# – Variação do *baudrate*

|  |  |  |  |
| --- | --- | --- | --- |
| **Baudrate (bits)** | **Tempo (s)** | **R (bits/tempo)** | **S (R/C)** |
| 9600 | 12,61722 | 6954,305307 | 0,724406803 |
| 19200 | 6,64387 | 13206,76052 | 0,68785211 |
| 38400 | 3,74516 | 23428,63856 | 0,610120796 |
| 57600 | 3,02182 | 29036,80563 | 0,504111209 |
| 115200 | 2,15521 | 40712,50597 | 0,35340717 |

Chart, line chart

Description automatically generated

L = 512

# Chart, line chart Description automatically generated– Variação do *timeout*

|  |  |  |  |
| --- | --- | --- | --- |
| ***timeout* (s)** | **Tempo (s)** | **R (bits/tempo)** | **S (R/C)** |
| 0 | 2,93484 | 29897,3709 | 3,114309468 |
| 1 | 4,38364 | 20016,24221 | 2,08502523 |
| 2 | 7,42891 | 11811,15399 | 1,230328541 |
| 3 | 9,84972 | 8908,273535 | 0,92794516 |
| 4 | 12,58759 | 6970,675086 | 0,726111988 |