Protocolo de Ligação de Dados

1º Trabalho Laboratorial

Afonso Gouveia Dias (up202006721@g.uporto.pt)
Ricardo Ribeiro Vieira(up202005091@g.uporto.pt)

November 2023

1 Sumário

Este primeiro trabalho laboratorial, no âmbito da cadeira de Redes de Computadores do 3.º ano da Licenciatura em Engenharia Informática e Computação, visou a implementação de um protocolo do nível de ligação de dados para a transmissão de ficheiros por meio de Portas série RS-232. Com este projeto foi possível consolidar a matéria lecionada e aprender a planear e prevenir contra adversidades que dificultam a transmissão dos dados que serão explicadas ao longo deste relatório.

2 Introdução

Como já foi referido, o principal objetivo do trabalho era implementar um protocolo do nível de ligação de dados conforme a especificação incluída no guião que nos foi fornecido, desenvolvendo uma aplicação simples de transferência de ficheiros com funcionalidades adequadas para aceder a ficheiros armazenados em disco (a correr no transmissor) e para armazenar um ficheiro em disco (a correr no recetor) e testar o protocolo com essa aplicação de transferência de ficheiros. Este relatório está dividido nas seguintes secções:

- Arquitetura: blocos funcionais e 'interfaces'.
- Estrutura do código: APIs, principais estruturas de dados, principais funções e a sua relação com a arquitetura.
- Casos de uso principais: identificação; sequências de chamada de funções.
- Protocolo de ligação lógica: identificação dos principais aspetos funcionais; descrição da estratégia de implementação destes aspetos com apresentação de extratos de código.
- Protocolo de aplicação: identificação dos principais aspetos funcionais; descrição da estratégia de implementação destes aspetos com apresentação de extratos de código.
- Validação: descrição dos testes efetuados com apresentação quantificada dos resultados, se possível.
- Eficiência do protocolo de ligação de dados: caraterização estatística da eficiência do protocolo, efetuada recorrendo a medidas sobre o código desenvolvido. A caraterização teórica de um protocolo 'Stop&Wait', que deverá ser empregue como termo de comparação, encontra-se descrita nos slides de Ligação Lógica das aulas teóricas.
- Conclusões: síntese da informação apresentada nas secções anteriores; reflexão sobre os objetivos de aprendizagem alcançados.

3 Arquitetura

3.1 Blocos Funcionais

O projeto está dividido em duas grandes partes: a aplicação (applicationlayer) e o protocolo de ligação de dados (linklayer).
A camada de ligação de dados, linklayer, é responsável pelo início (llopen) e fim (llclose) da ligação, pela criação e envio de tramas (llwrite) e pela validação das tramas recebidas (llread). Por outro lado, a camada de aplicação, applicationlayer, é responsável, sobretudo, pela definição dos pacotes de controlo e de informação e utiliza chamadas às funções da linklayer para a transferência e receção dos pacotes de dados do ficheiro a transferir.

3.2 Interface

Para executar o programa basta adaptar o makefile para ter o número correto nas portas série, abrir um terminal em cada computador e correr o computador com o ficheiro em modo TX (transmissor) make run_tx e o computador que vai receber em modo RX (recetor) com make run_rx.
Para confirmar se a transferência foi bem efetuada corre-se make check_files. Além disso, se quisermos testar o programa com um só computador, pudemos usar sudo make run_cable para correr o cabo virtual fornecido.

4 Estrutura do código

4.1 ApplicationLayer

As funções implementadas:

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

// Funcao auxiliar para calcular os bytes que um numero "n" ocupa
unsigned int bytesToRepresent(long int n);
```

4.2 LinkLayer

No caso desta camada, além das funções, temos algumas estruturas de dados:

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

typedef struct
```

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

typedef enum
{
    CMD_TX, // Commands sent by the transmitter
    CMD_RX, // Commands sent by the receiver
    ANS_TX, // Replies sent by the transmitter
    ANS_RX, // Replies sent by the receiver
} AddressFieldType;

typedef enum {
    Start,
    FlagRCV,
    ARCV,
    CRCV,
    BccOK,
    Data,
    Destuffing
} stateNames;

typedef struct
{
    bool stuffed;
    unsigned char byte1;
    unsigned char byte2;

} StuffingAux;
```

A estrutura LinkLayerRole identifica se estamos perante o transmissor ou recetor, na de LinkLayer são definidos os parâmetros associados à transferência de dados, com AddressFieldType temos possibilidades de endereços, stateNames identifica o estado de leitura de trama em que o programa está e, por último, StuffingAux ajuda-nos a agrupar bytes relacionados com stuffing e destuffing.

As funções implementadas:

```
// Abrir a ligacao com os parametros definidos numa struct LinkLayer
int llopen(LinkLayer connectionParameters);

// Enviar os dados em buf
int llwrite(const unsigned char *buf, int bufSize);
```

```
// Funcao auxiliar ao llwrite que envia um frame e espera
// pela resposta do recetor
int llwriteSendFrame(unsigned char *frame, int frameSize);

// Receber e analisar os dados em packet.
int llread(unsigned char *packet);

// Fechar a ligacao aberta no inicio do programa
int llclose(int showStatistics, LinkLayer connectionParameters);

// Dar escape a uma flag ou um escape em byte
StuffingAux stuffByte(unsigned char byte);

// Cria um frame de resposta para o recetor enviar ao transmissor para o
// mesmo saber se a info frame foi bem recebida ou rejeitada
int sendRxResponse(unsigned char C);

// Lida com alarm interrupts
void alarmHandler(int signal);
```

5 Casos de uso principais

Como já anteriormente explicado, o programa é corrido como transmissor num computador e recetor noutro. Sendo as chamadas e a sua ordem ligeiramente diferente.
Na applicationlayer é chamado o llopen(), usada para o iniciar a transmissão entre o transmissor e o recetor, através da troca de pacotes de controlo e conexão com a porta série.

5.1 Transmissor

1. Criação do start control packet.
2. llwrite(), que envia este mesmo start control packet.
3. Criação de data packets e envio através de llwrite enquanto ainda há data para enviar.
4. Modificação do control packet criado no início para ser lido como end control packet.
5. llwrite(), que envia este mesmo end control packet.

5.2 Recetor

1. llread(), que gere e valida a receção do start control packet.
2. Criação e abertura do ficheiro recebido além da criação do nome.
3. llread, enquanto não recebemos o end control packet e escrever a data para o ficheiro criado.

E por último, llclose(), usado para terminar a ligação entre o transmissor e o recetor.

Notas:
llwrite trata de qualquer stuffing necessário.
llread trata de qualquer destuffing necessário.

6 Protocolo de ligação lógica

A camada de ligação de dados desempenha um papel fundamental na comunicação entre o transmissor e o recetor, interagindo diretamente com a Porta Série. Nesse processo, empregamos o protocolo Stop-and-wait tanto para iniciar e encerrar a conexão quanto para o envio de tramas de supervisão e informação.

A função llopen é responsável por iniciar a ligação. Primeiramente, após a abertura e configuração da porta série, o transmissor envia uma trama de supervisão SET e aguarda a resposta do recetor na forma de uma trama de supervisão UA. Quando o recetor recebe o SET, ele responde com UA. Se o transmissor receber a trama UA, isso indica que a conexão foi estabelecida com sucesso. Uma vez que a conexão esteja estabelecida, o transmissor inicia a transmissão de informações que serão recebidas e processadas pelo recetor.

A função llwrite é responsável pelo envio de informações. Essa função recebe um pacote, que pode ser de controle ou de dados, e aplica a técnica de byte stuffing para evitar conflitos com bytes de dados que possam coincidir com as 'flags'/escapes da trama. Em seguida, o pacote é transformado em uma trama de informação (framing) e é enviado ao recetor, aguardando-se a resposta dele. Se a trama for rejeitada, o envio é repetido até ser aceite ou até atingir o número máximo de tentativas permitidas. Cada tentativa de envio é limitada por um tempo, após o qual ocorre um timeout.

A função llread é responsável pela leitura de informações. Ela opera lendo os dados recebidos pela porta série e avaliando a sua integridade. Primeiramente, realiza o processo de destuffing no campo de dados da trama e, em seguida, verifica a validade dos valores BCC1 e BCC2. Essa verificação tem o propósito de identificar qualquer erro que possa ter ocorrido durante a transmissão.

O encerramento da ligação é efetuado através da função llclose. Essa função é chamada quando a transferência dos pacotes de dados está completa. O transmissor envia uma trama de supervisão DISC e aguarda a resposta do recetor, que também envia uma trama DISC em resposta. Isso marca o término da

operação. Quando o transmissor recebe o DISC, ele responde com uma trama UA e interrompe a ligação.

7 Protocolo de aplicação

O protocolo de aplicação é a camada onde fazemos a interação com o ficheiro a ser transferido. Ela começa por incluir os cabeçalhos necessários e define funções úteis, como 'bytesToRepresent', que calcula o número de bytes necessários para representar um número inteiro longo em binário. A função 'applicationLayer' serve como ponto central do programa, recebendo parâmetros que descrevem a porta serial, o papel (transmissor ou recetor), a taxa de transmissão, entre outros. Após inicializar os parâmetros de conexão, a função chama 'llopen' para estabelecer a conexão da camada de ligação.Dependendo da função (transmissor ou recetor), o código executa uma lógica diferente. No papel de transmissor (LlTx), abre um ficheiro, calcula o seu tamanho e envia um pacote de controlo inicial, seguido da transmissão do conteúdo do ficheiro em partes. Finalmente, é enviado um pacote de controlo final. Na função de recetor (LlRx), o código aguarda o pacote de controlo de início, cria um ficheiro de saída com um nome modificado e lê os pacotes de dados, escrevendo o conteúdo no ficheiro de saída. A leitura continua até ser recebido um pacote de controlo final. Por fim, a função 'llclose' é chamada para encerrar a conexão da camada de 'link' e potencialmente exibir estatísticas de transmissão.

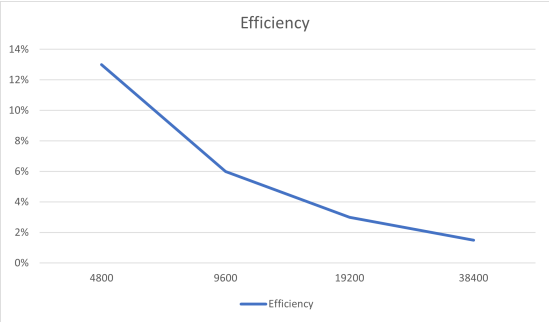
8 Validação

Durante a realização do trabalho para garantir a funcionalidade correta do sistema foram realizados os seguintes testes:

1. Ligar apenas o transmissor e verificar o funcionamento do alarme.
2. Introdução de ruído na transferência e verificar se o ficheiro chegava ao recetor corretamente.
3. Desligar do cabo de ligação e verificar se ao voltar a conectar o cabo a ligação continua sem problemas e o ficheiro chega corretamente ao recetor.

9 Eficiência do protocolo de ligação de dados

A eficiência do protocolo de ligação de dados seria calculada variando 3 parâmetros separadamente (Baudrate, tamanho da trama e o FER(Frame Error Rate). Apenas foi calculado a variação da baurate usando o ficheiro cable.c



O gráfico a cima representa o aumento da eficiência com uma baudrate menor e a diminuição da eficiência com maior baudrate. O gráfico foi calculado usando a fórmula S = R/C, onde R representa o débito recebido em bits/s e o C representa a capacidade da ligação em bits/s. C simplesmente era a baudrate usada e o C era o tamanho do ficheiro a dividir pelo tempo que demorou a transferir esse mesmo ficheiro. Para ter eficiência de 100%, teríamos de estar sempre a transmitir informação sem parar, mas não é o caso por duas razões:

1. O protocolo que implementamos envolve bastantes esperas.
2. Os headers/tailers de cada frame existem. Isto quer dizer que uma parte da capacidade está a ser usada para bits "inúteis".

10 Conclusões

O protocolo de ligação de dados desempenhou um papel crucial na compreensão dos conceitos abordados nas aulas teóricas. Este protocolo consistia na LinkLayer, que lidava com a comunicação através da porta série e geria as tramas de informação, e na ApplicationLayer, que interagia diretamente com o ficheiro a ser transferido. Através deste projeto, conseguimos assimilar os princípios do byte stuffing, framing e o funcionamento do protocolo Stop-and-Wait, incluindo a sua capacidade de deteção e tratamento de erros.
Gostávamos, por último, de adicionar uma pequena nota relativa à apresentação. Nesta mesma detetámos um erro após tentarmos correr apenas o terminal do transmissor. Apesar de a ligação não ser estabelecida, o programa continuava, sendo que apenas na primeira chamada do llwrite lidava com esse problema após os vários alarmes. Este erro era facilmente corrigido modificando a linha

```
llopen(connectionParameters);
```

por

```
if (llopen(connectionParameters) == -1){
  exit(-1);
      }
```

11 Anexo I - Código fonte.

11.1 application_layer.h

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

#ifndef _APPLICATION_LAYER_H_
#define _APPLICATION_LAYER_H_

#include <math.h>

// SHOW FINAL STATISTICS

#define SHOW_STATISTICS TRUE

// CONTROL PACKET

#define CTRL_START 2
#define CTRL_END 3
#define T_SIZE 0
#define T_NAME 1

// 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);

// Auxiliary function to calculate how many bytes a number takes up
// Returns the number of bytes of n
unsigned int bytesToRepresent(long int n);

#endif // _APPLICATION_LAYER_H_
```

11.2 application_layer.c

```
// Application layer protocol implementation
```

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

unsigned int bytesToRepresent(long int n)
{
    return ceil(log2f((float)n) / 8.0); // log2 get the highest bit set, and division by 8 f
}

void applicationLayer(const char *serialPort, const char *role, int baudRate,
                      int nTries, int timeout, const char *filename)
{
    LinkLayer connectionParameters;
    strcpy(connectionParameters.serialPort, serialPort);
    connectionParameters.role = strcmp(role, "tx") ? LlRx : LlTx;
    connectionParameters.baudRate = baudRate;
    connectionParameters.nRetransmissions = nTries;
    connectionParameters.timeout = timeout;

    llopen(connectionParameters);

    // Send Start Control Packet
    switch (connectionParameters.role)
    {
    case LlTx:
    {
        // Create start control packet
        FILE *file = fopen(filename, "rb");
        if (file == NULL)
        {
            perror("[application_layer] File not found\n");
            exit(-1);
        }

        fseek(file, 0L, SEEK_END);
        long int fileSize = ftell(file); // File size in bytes
        fseek(file, 0L, SEEK_SET);       // Reset file descriptor to start of file

        unsigned char L1 = bytesToRepresent(fileSize);
        unsigned char L2 = strlen(filename);
        int ctrlsize = 3 + L1 + 2 + L2;

        unsigned char control[ctrlsize];

        control[0] = (unsigned char)CTRL_START;
        control[1] = (unsigned char)T_SIZE;
```

```
control[2] = L1;
unsigned int idx = 3;

for (int i = L1; i > 0; i--, idx++)
{
    control[idx] = (fileSize >> (8 * (i - 1))) & 0xff;
}
control[idx++] = T_NAME;
control[idx++] = L2;

for (int i = 0; i < L2; i++, idx++)
{
    control[idx] = filename[i];
}

if (llwrite(control, ctrlsize) == -1)
{
    printf("[application_layer] Error: Transmitter obtained no response after alarms
    exit(-1);
}
printf("[application_layer] Wrote start control packet\n\n");

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

long int leftoverBytes = fileSize;

while (leftoverBytes > 0)
{
    unsigned int numBytes = leftoverBytes < MAX_PAYLOAD_SIZE - 3 ? leftoverBytes : M
    unsigned char data[numBytes];

    data[0] = 1;
    data[1] = (numBytes >> 8) & 0xff;
    data[2] = numBytes & 0xff;
    memcpy(data + 3, fileContent, numBytes);

    if (llwrite(data, 3 + numBytes) == -1)
    {
        printf("[application_layer] Error: Transmitter obtained no response after al
        fclose(file);
        exit(-1);
    }

    printf("[application_layer] Wrote frame of %d byte(s)\n\n", numBytes);
    leftoverBytes -= numBytes;
```

```
            fileContent += numBytes;
        }

        control[0] = 3;
        if (llwrite(control, ctrlsize) == -1)
        {
            printf("[application_layer] Error: Transmitter obtained no response after alarms
            fclose(file);
            exit(-1);
        }
        printf("[application_layer] Wrote end control packet\n\n");
        fclose(file);
        break;
    }
    case LlRx:
    {
        unsigned char packet[MAX_PAYLOAD_SIZE];
        while (llread(packet) == -1) // Read start control packet
            ;
        printf("[application_layer] Read start control packet\n");

        unsigned int numFieldBytes = packet[2];
        long int fileSize = 0;

        for (int i = 3, exp = numFieldBytes - 1; i < 3 + numFieldBytes; i++, exp--)
        {
            fileSize += packet[i] << (8 * exp);
        }

        unsigned int numNameBytes = packet[2 + numFieldBytes + 2];
        unsigned char receiverFilename[numNameBytes + 10]; // Initial filename + "-received'
        unsigned char addon[9] = "-received";

        // Add "-received" to filename
        for (int i = numFieldBytes + 5, j = 0; i <= numFieldBytes + 5 + numNameBytes; i++, j
        {
            if (packet[i] == '.')
            {
                for (int k = 0; k < 9; k++)
                {
                    receiverFilename[j++] = addon[k];
                }
            }
            receiverFilename[j] = packet[i];
        }
        receiverFilename[numNameBytes + 9] = '\0';
```

```
        unsigned int leftoverBytes = fileSize;
        FILE *outputFile = fopen((char *)receiverFilename, "wb+");

        while (packet[0] != 3) // While we haven't received a control end packet
        {

            while (llread(packet) == -1)
                ;
            numFieldBytes = 256 * packet[1] + packet[2];
            leftoverBytes -= numFieldBytes;

            if (packet[0] != 3) // If the packet is not a stop control packet, write the dat
            {
                printf("[application_layer] Read packet\n");
                unsigned char *dataStart = packet + 3;
                fwrite(dataStart, sizeof(unsigned char), numFieldBytes, outputFile);
            }
        }
        printf("[application_layer] Read end control packet\n\n");
        break;
    }
    }

    llclose(SHOW_STATISTICS, connectionParameters);

}
```

11.3 link_layer.h

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

#ifndef _LINK_LAYER_H_
#define _LINK_LAYER_H_

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

```
#include <signal.h>
#include <stdbool.h>
#include <sys/time.h>

typedef enum
{
    LlTx,
    LlRx,
} LinkLayerRole;

typedef enum
{
    CMD_TX, // Commands sent by the transmitter
    CMD_RX, // Commands sent by the receiver
    ANS_TX, // Replies sent by the transmitter
    ANS_RX, // Replies sent by the receiver

} AddressFieldType;

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

typedef struct
{
    bool stuffed;
    unsigned char byte1;
    unsigned char byte2;

} StuffingAux;

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

// 1000 - 3 from PA means each image chunk will have 997 bytes
#define MAX_PAYLOAD_SIZE 1000

// MISC
#define FALSE 0
#define TRUE 1
#define FLAG 0x7E
```

```
#define ESC 0x7D

// ADDRESS FIELD TYPES
#define CMD_TX 0x03
#define CMD_RX 0x01
#define ANS_TX 0x01
#define ANS_RX 0x03

// SUPERVISION FRAMES
#define C_SET 0x03
#define C_UA 0x07
#define C_RR0 0x05
#define C_RR1 0x85
#define C_REJ0 0x01
#define C_REJ1 0x81
#define C_DISC 0x0B

// INFORMATION FRAMES
#define C_FRAME0 0x00
#define C_FRAME1 0x40

// STATE MACHINES

typedef enum {
    Start,
    FlagRCV,
    ARCV,
    CRCV,
    BccOK,
    Data,
    Destuffing
} stateNames;

// 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);

// Auxiliary function that sends a frame and waits for the appropriate RR response from the
// Returns 1 if it got a succesful response, 0 if the frame got rejected or -1 if no respons
int llwriteSendFrame(unsigned char *frame, int frameSize);

// 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, LinkLayer connectionParameters);

// Escapes flags and escape characters in byte
// Returns a struct containg a bool, that indicates if the byte required stuffing or not, ar
StuffingAux stuffByte(unsigned char byte);

//Creates a supervision frame for receiver replies to information frames.
// Returns the number of bytes written (5) in success or -1 if theres any errors
int sendRxResponse(unsigned char C);

// Handles alarm interrupts
void alarmHandler(int signal);

#endif // _LINK_LAYER_H_
```

11.4 link_layer.c

```
// Link layer protocol implementation
#include "link_layer.h"

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

volatile int STOP = FALSE;
int alarmEnabled = FALSE;
int alarmCount = 0;

int fd;
unsigned char chr[1] = {0};

struct termios oldtio;
struct termios newtio;

bool frameCountTx = 0;
bool frameCountRx = 0;

int tries;
int timeout;
stateNames state = Start;
```

```
struct timeval stop, start;

// Alarm function handler
void alarmHandler(int signal)
{
    alarmEnabled = FALSE;
    alarmCount++;

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

int sendRxResponse(unsigned char C)
{
    switch (C)
    {
    case C_RR0:
        printf("Sent response RR0\n");
        break;
    case C_RR1:
        printf("Sent response RR1\n");
        break;
    case C_REJ0:
        printf("Sent response REJ0\n");
        break;
    case C_REJ1:
        printf("Sent response REJ1\n");
        break;
    }

    unsigned char frame[5] = {FLAG, ANS_RX, C, ANS_RX ^ C, FLAG};
    return write(fd, frame, 5);
}


////////////////////////////////////////////////
// LLOPEN
////////////////////////////////////////////////
int llopen(LinkLayer connectionParameters)
{
    gettimeofday(&start, NULL);
    // Open serial port device for reading and writing, and not as controlling tty
    // because we don't want to get killed if linenoise sends CTRL-C.
    fd = open(connectionParameters.serialPort, O_RDWR | O_NOCTTY);

    if (fd < 0)
```

```
{
    perror(connectionParameters.serialPort);
    exit(-1);
}

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

// Clear struct for new port settings
memset(&newtio, 0, sizeof(newtio));

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

// Set input mode (non-canonical, no echo,...)
newtio.c_lflag = 0;
newtio.c_cc[VTIME] = 0; // Inter-character timer unused
newtio.c_cc[VMIN] = 0;  // the read() func will return immediately, with either the numb

// VTIME e VMIN should be changed in order to protect with a
// timeout the reception of the following character(s)

tcflush(fd, TCIOFLUSH);

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

printf("New termios structure set\n");

tries = connectionParameters.nRetransmissions;
timeout = connectionParameters.timeout;

if(connectionParameters.role == LlTx){

    unsigned char buf[5] = {FLAG, CMD_TX, C_SET, CMD_TX ^ C_SET, FLAG};

    // Set alarm function handler
```

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


alarmEnabled = FALSE;
alarmCount = 0;

while (alarmCount <= tries)
{
    if (alarmEnabled == FALSE)
    {
        alarm(timeout); // Set alarm to be triggered in timeout seconds
        alarmEnabled = TRUE;

        int write_bytes = write(fd, buf, 5);
        sleep(1);
        printf("%d bytes written\n", write_bytes);
    }

    // Returns after a char has been input
    int read_bytes = read(fd, chr, 1);
    if (read_bytes)
    {
        switch (state)
        {
        case Start:
        {
            if (chr[0] == FLAG)
                state = FlagRCV;
            break;
        }
        case FlagRCV:
        {
            if (chr[0] == ANS_RX) // This is specific state machine for receiving UA
                state = ARCV;
            else if (chr[0] != FLAG)
                state = Start;
            break;
        }
        case ARCV:
        {
            if (chr[0] == C_UA)
                state = CRCV;
            else if (chr[0] == FLAG)
                state = FlagRCV;
            else
                state = Start;
```

```
                break;
            }
            case CRCV:
            {
                if (chr[0] == (ANS_RX ^ C_UA)) // In a more general state machine, where
                    state = BccOK;
                else if (chr[0] == FLAG)
                    state = FlagRCV;
                else
                    state = Start;
                break;
            }

            case BccOK:
            {
                if (chr[0] == FLAG)
                {
                    printf("Read UA\n\n");

                    alarm(0);
                    return 1;
                }
                else
                    state = Start;
                break;
            }
            }
        }
    }
}
else if(connectionParameters.role == LlRx){
        // Loop for input
    chr[0] = 0;
    state = Start;
    STOP = FALSE;

    while (STOP == FALSE)
    {
        // Returns after a char has been input
        int read_bytes = read(fd, chr, 1);

        if (read_bytes)
        {
            switch (state)
            {
            case Start:
```

```
{
    if (chr[0] == FLAG)
        state = FlagRCV;
    break;
}
case FlagRCV:
{
    if (chr[0] == CMD_TX)
        state = ARCV;
    else if (chr[0] != FLAG)
        state = Start;
    break;
}
case ARCV:
{
    if (chr[0] == CMD_TX)
        state = CRCV;
    else if (chr[0] == FLAG)
        state = FlagRCV;
    else
        state = Start;
    break;
}
case CRCV:
{
    if (chr[0] == (CMD_TX ^ C_SET))
        state = BccOK;
    else if (chr[0] == FLAG)
        state = FlagRCV;
    else
        state = Start;
    break;
}
case BccOK:
{
    if (chr[0] == FLAG)
    {
        printf("Read SET\n");

        unsigned char UA[5] = {FLAG, ANS_RX, C_UA, ANS_RX ^ C_UA, FLAG};
        int bytes = write(fd, UA, 5);
        sleep(1);
        printf(" %d UA bytes written\n\n", bytes);

        STOP = TRUE;
    }
```

```
                    else
                    {
                        state = Start;
                    }
                    break;
                }
                }
            }
        }
    }
    return -1;
}

int llwriteSendFrame(unsigned char *frame, int frameSize)
{
    unsigned char expectedResponse;
    unsigned char rejection;
    if(frameCountTx == 1){
        expectedResponse = C_RR0;
        rejection = C_REJ1;
    }
    else {
        expectedResponse = C_RR1;
        rejection = C_REJ0;
    }

    // Set alarm function handler
    (void)signal(SIGALRM, alarmHandler);

    chr[0] = 0;
    state = Start;
    alarmEnabled = FALSE;
    alarmCount = 0;

    while (alarmCount <= tries)
    {
        if (alarmEnabled == FALSE)
        {
            alarm(timeout); // Set alarm to be triggered in timeout seconds
            alarmEnabled = TRUE;

            int write_bytes = write(fd, frame, frameSize);
            sleep(1);

        }
```

```
// Returns after a char has been input
int read_bytes = read(fd, chr, 1);
if (read_bytes)
{
    switch (state)
    {
    case Start:
    {
        if (chr[0] == FLAG)
            state = FlagRCV;
        break;
    }
    case FlagRCV:
    {
        if (chr[0] == ANS_RX)
            state = ARCV;
        else if (chr[0] != FLAG)
            state = Start;
        break;
    }
    case ARCV:
    {
        if (chr[0] == expectedResponse)
            state = CRCV;
        else if (chr[0] == FLAG)
            state = FlagRCV;
        else if (chr[0] == rejection)
        {
            printf("Read RJ\n");
            alarm(0);
            return 0;
        }
        else
            state = Start;
        break;
    }
    case CRCV:
    {
        if (chr[0] == (ANS_RX ^ expectedResponse))
            state = BccOK;
        else if (chr[0] == FLAG)
            state = FlagRCV;
        else
            state = Start;
        break;
    }
```

```
            case BccOK:
            {
                if (chr[0] == FLAG)
                {
                    printf("Read RR%d\n", !frameCountTx);
                    alarm(0);
                    frameCountTx = !frameCountTx;
                    return 1;
                }
                else
                    state = Start;
                break;
            }
            }
        }
    }
    printf("Obtained no appropriate response after numRetransmission alarms\n");
    alarm(0);
    return -1;
}

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

    unsigned char infFrame[MAX_PAYLOAD_SIZE * 2];
    unsigned char controlField;

    if(frameCountTx == 1){
        controlField = C_FRAME1;
    }
    else{
        controlField = C_FRAME0;
    }

    unsigned char header[4] = {FLAG,
                               CMD_TX,
                               controlField,
                               CMD_TX ^ controlField};

    unsigned result_idx = 4; // Next idx on which to write data
    memcpy(infFrame, header, 4);

    unsigned char bcc2 = buf[0];
```

```
// Stuffing Data
StuffingAux stuffData;

for (int i = 0; i < bufSize; i++)
{
    stuffData = stuffByte(buf[i]);

    if (stuffData.stuffed)
    {
        infFrame[result_idx++] = stuffData.byte1;
        infFrame[result_idx] = stuffData.byte2;
    }
    else
    {
        infFrame[result_idx] = buf[i];
    }
    result_idx++;

    if (i)
    { // If it is not the first chunk of data, XOR it with the rest
        bcc2 = bcc2 ^ buf[i];
    }
}

StuffingAux stuffBCC2 = stuffByte(bcc2);

if (stuffBCC2.stuffed)
{
    infFrame[result_idx++] = stuffBCC2.byte1;
    infFrame[result_idx++] = stuffBCC2.byte2;
}
else
{
    infFrame[result_idx++] = bcc2;
}

infFrame[result_idx] = FLAG;

int result = llwriteSendFrame(infFrame, 2 * bufSize + 6);

while (result == 0)
{
    result = llwriteSendFrame(infFrame, 2 * bufSize + 6); // Resend frame while REJs are
}
return result == 1 ? bufSize + 6 : -1;
```

```
}

////////////////////////////////////////////////
// LLREAD
////////////////////////////////////////////////
int llread(unsigned char *packet)
{
    chr[0] = 0;
    state = Start;
    STOP = FALSE;

    bool receivedFrame;

    unsigned int packetIdx = 0;
    unsigned char bcc2;

    while (STOP == FALSE)
    {
        // Returns after a char has been input
        int read_bytes = read(fd, chr, 1);
        if (read_bytes)
        {
            switch (state)
            {
            case Start:
            {
                if (chr[0] == FLAG)
                    state = FlagRCV;
                break;
            }
            case FlagRCV:
            {
                if (chr[0] == CMD_TX)
                    state = ARCV;
                else if (chr[0] != FLAG)
                    state = Start;
                break;
            }
            case ARCV:
            {
                if (chr[0] == C_FRAME0)
                {
                    receivedFrame = 0;
                    state = CRCV;
                }
                else if (chr[0] == C_FRAME1)
```

```
    {
        receivedFrame = 1;
        state = CRCV;
    }
    else if (chr[0] == FLAG)
        state = FlagRCV;
    else
        state = Start;
    break;
}
case CRCV:
{
    unsigned int receivedC = receivedFrame ? 0x40 : 0x00;
    if (chr[0] == (CMD_TX ^ receivedC))
        state = Data;
    else if (chr[0] == FLAG)
        state = FlagRCV;
    else
        state = Start;
    break;
}
case Data:
{
    if (chr[0] == ESC)
        state = Destuffing;
    else if (chr[0] == FLAG)
    {
        printf("Read whole input\n");
        unsigned char receivedBcc2 = packet[packetIdx - 1];
        bcc2 = packet[0];

        packet[packetIdx - 1] = '\0'; // Remove read bcc2 because the applicatio

        for (int i = 1; i < packetIdx - 1; i++) // Start in index 1 because inde
        {
            bcc2 = bcc2 ^ packet[i];
        }

        if (bcc2 == receivedBcc2) // Frame data has NO errors
        {
            if (receivedFrame == frameCountRx)
            {
                frameCountRx = !frameCountRx;
                if(frameCountRx==1){ // Updated value with what frame receiver e
                    sendRxResponse(C_RR1);
                }
```

```
                else{
                    sendRxResponse(C_RR0);
                }
                printf("Read expected frame\n");
                return packetIdx;
            }
            else // Duplicate Frame
            {
                if(frameCountRx==1){
                    sendRxResponse(C_RR1);
                }
                else{
                    sendRxResponse(C_RR0);
                } // Receiver is still expecting the same frame, because it just
                printf("Read duplicate frame\n");
                return -1;
            }
        }
        else // Frame data HAS errors
        {
            if (receivedFrame == frameCountRx)
            {
                if(frameCountRx==1){
                    sendRxResponse(C_REJ1);
                }
                else{
                    sendRxResponse(C_REJ0);
                } // Receiver is still expecting the same frame, because it just
                printf("Read expected frame with errors\n");
                return -1;
            }
            else // Duplicate Frame with errors
            {
                if(frameCountRx==1){
                    sendRxResponse(C_RR1);
                }
                else{
                    sendRxResponse(C_RR0);
                }
                printf("Read duplicate frame with errors\n");
                return -1;
            }
        }
    }
    else
        packet[packetIdx++] = chr[0];
```

```
                break;
            }
            case Destuffing:
            {
                if (chr[0] == 0x5E)       // in case what was stuffed was the equivalent of
                {
                    packet[packetIdx++] = 0x7E;
                    state = Data;
                }
                else if (chr[0] == 0x5D)  // in case what was stuffed was the equivalent of
                {
                    packet[packetIdx++] = 0x7D;
                    state = Data;
                }
                else                      // in case there was something wrong with stuffing
                {
                    printf("An ESC wasn't stuffed\n");
                    state = Data;
                }
                break;
            }
            }
        }
    }
    return -1;
}

////////////////////////////////////////////////
// LLCLOSE
////////////////////////////////////////////////

int llclose(int showStatistics, LinkLayer connectionParameters)
{
    if(connectionParameters.role == LlTx){

        unsigned char buf[5] = {FLAG, CMD_TX, C_DISC, CMD_TX ^ C_DISC, FLAG};

        // Set alarm function handler
        (void)signal(SIGALRM, alarmHandler);

        chr[0] = 0;
        state = Start;
        alarmEnabled = FALSE;
        alarmCount = 0;

        while (alarmCount <= tries)
```

```
{
    if (alarmEnabled == FALSE)
    {
        alarm(timeout); // Set alarm to be triggered in timeout seconds
        alarmEnabled = TRUE;

        int write_bytes = write(fd, buf, 5);
        sleep(1);
        printf(" %d bytes written\n", write_bytes);
    }

    // Returns after a char has been input
    int read_bytes = read(fd, chr, 1);
    if (read_bytes)
    {
        switch (state)
        {
        case Start:
            if (chr[0] == FLAG)
                state = FlagRCV;
            break;
        case FlagRCV:
            if (chr[0] == CMD_RX)
                state = ARCV;
            else if (chr[0] != FLAG)
                state = Start;
            break;
        case ARCV:
            if (chr[0] == C_DISC)
                state = CRCV;
            else if (chr[0] == FLAG)
                state = FlagRCV;
            else
                state = Start;
            break;
        case CRCV:
            if (chr[0] == (C_DISC ^ CMD_RX))
                state = BccOK;
            else if (chr[0] == FLAG)
                state = FlagRCV;
            else
                state = Start;
            break;
        case BccOK:
            if (chr[0] == FLAG)
            {
```

```
                    printf("Read C_DISC\n");

                    unsigned char UA[5] = {FLAG, ANS_TX, C_UA, ANS_TX ^ C_UA, FLAG};
                    int bytes = write(fd, UA, 5);
                    sleep(1);
                    printf(" %d UA bytes written\n", bytes);

                    alarm(0);
                    return 1;
                }
                else
                    state = Start;
                break;
            }
        }
    }
}
else if(connectionParameters.role == LlRx){
    state = Start;
    chr[0] = 0;

    STOP = FALSE;

    while (STOP == FALSE)
    {
        // Returns after a char has been input
        int read_bytes = read(fd, chr, 1);
        if (read_bytes)
        {
            switch (state)
            {
            case Start:
                if (chr[0] == FLAG)
                    state = FlagRCV;
                break;
            case FlagRCV:
                if (chr[0] == CMD_TX)
                    state = ARCV;
                else if (chr[0] != FLAG)
                    state = Start;
                break;
            case ARCV:
                if (chr[0] == C_DISC)
                    state = CRCV;
                else if (chr[0] == FLAG)
                    state = FlagRCV;
```

```
            else
                state = Start;
            break;
        case CRCV:
            if (chr[0] == (C_DISC ^ CMD_TX))
                state = BccOK;
            else if (chr[0] == FLAG)
                state = FlagRCV;
            else
                state = Start;
            break;
        case BccOK:
            if (chr[0] == FLAG)
            {
                printf("Read C_DISC\n");
                STOP = true;
            }
            else
                state = Start;
        }
    }
}

unsigned char buf[5] = {FLAG, CMD_RX, C_DISC, CMD_RX ^ C_DISC, FLAG};

(void)signal(SIGALRM, alarmHandler);

chr[0] = 0;
state = Start;
alarmEnabled = FALSE;
alarmCount = 0;

// Write C_C_DISC and wait for UA from transmitter
while (alarmCount <= tries)
{
    if (alarmEnabled == FALSE)
    {
        alarm(timeout); // Set alarm to be triggered in timeout seconds
        alarmEnabled = TRUE;

        int write_bytes = write(fd, buf, 5);
        sleep(1);
        printf(" %d bytes written\n", write_bytes);
    }

    // Returns after a char has been input
```

```
int read_bytes = read(fd, chr, 1);
if (read_bytes)
{
    switch (state)
    {
    case Start:
    {
        if (chr[0] == FLAG)
            state = FlagRCV;
        break;
    }
    case FlagRCV:
    {
        if (chr[0] == ANS_TX) // This is specific state machine for receiving UA
            state = ARCV;
        else if (chr[0] != FLAG)
            state = Start;
        break;
    }
    case ARCV:
    {
        if (chr[0] == C_UA)
            state = CRCV;
        else if (chr[0] == FLAG)
            state = FlagRCV;
        else
            state = Start;
        break;
    }
    case CRCV:
    {
        if (chr[0] == (ANS_TX ^ C_UA)) // In a more general state machine, where
            state = BccOK;
        else if (chr[0] == FLAG)
            state = FlagRCV;
        else
            state = Start;
        break;
    }

    case BccOK:
    {
        if (chr[0] == FLAG)
        {
            printf("Read UA\n");
            alarm(0);
```

```
                        gettimeofday(&stop, NULL);

                        printf("took %lu us to transfer the file\n", (stop.tv_sec - start.tv
                        return 1;
                    }
                    else
                        state = Start;
                    break;
                }
                }
            }
        }
    }

    // Restore the old port settings
    if (tcsetattr(fd, TCSANOW, &oldtio) == -1)
    {
        perror("tcsetattr");
        exit(-1);
    }

    close(fd);


    return 1;
}

StuffingAux stuffByte(unsigned char byte)
{
    StuffingAux result;

    switch (byte)
    {
    case FLAG:
    {
        result.stuffed = true;
        result.byte1 = ESC;
        result.byte2 = 0x5E;
        return result;
    }
    case ESC:
    {
        result.stuffed = true;
        result.byte1 = ESC;
        result.byte2 = 0x5D;
        return result;
```

```
    }
    default:
    {
        result.stuffed = FALSE;
        result.byte1 = byte;
        return result;
    }
    }
}
```