# Protocolo de Ligação de Dados

1º Trabalho Laboratorial


## Redes de Computadores
## Turma 7


- Diogo Silveira Viana – up202108803
- Gonçalo de Castilho Serra Alves Martins – up202108707

# Sumário

Este relatório foi realizado no âmbito da unidade curricular de Redes de Computadores (RCOM) do 3º ano de licenciatura em engenharia informática e de computação (LEIC), tendo este trabalho laboratorial, como objetivo a implementação de um protocolo de comunicação de dados para a transmissão de ficheiros, entre dois computadores, utilizando a Porta Série RS-232.

Ao longo da elaboração deste projeto foi-nos possível utilizar e consolidar a matéria lecionada nesta unidade curricular durante as aulas teóricas para a implementação do protocolo em questão, consolidando também, desta forma, o funcionamento da estratégia *Stop-and-Wait*.

Este relatório tem apenas como objetivo detalhar a nossa implementação do que nos foi proposto no guião do trabalho laboratorial, bem como a explicação dos conceitos teóricos que foram utilizados.

# Introdução

O trabalho teve como objetivo, o desenvolvimento e teste de um protocolo de ligação de dados que, por sua vez, visa fornecer um serviço de comunicação de dados fiável entre dois computadores ligados, neste caso, por uma porta série, de forma a possibilitar a transferência de um ficheiro de um para o outro, tudo isto de acordo com as especificações fornecidas no guião deste trabalho laboratorial.

Visto isto, o relatório irá ter a seguinte estrutura:

- **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.
- **Eficiência do Protocolo de ligação de dados:** caracterização estatística da eficiência do protocolo, efetuada recorrendo a medidas sobre o código desenvolvido.
- **Conclusões:** síntese da informação apresentada nas secções anteriores, reflexão sobre os objetivos de aprendizagem alcançados.

# Arquitetura

## Blocos Funcionais

O projeto foi desenvolvido em duas camadas principais: a *ApplicationLayer* e a *LinkLayer*.

A *ApplicationLayer* é implementada nos ficheiros application_layer.h e application_layer.c. Esta camada utiliza a API da *LinkLayer* para o envio e receção de pacotes de dados que, por sua vez, constituem um ficheiro. Como tal, das duas camadas, é a mais

próxima ao utilizador e é possível definir, na mesma, o tamanho das tramas de informação, a velocidade da transferência e o número máximo de retransmissões.

A *LinkLayer* é implementada nos ficheiros link_layer.h e link_layer.c. Esta camada é, então, responsável pelo estabelecimento e terminação da ligação, criação e envio de tramas de dados através da porta série e, por fim, da validação das tramas recebidas e do envio de mensagens de erro, no caso de algum ter ocorrido durante a transmissão, consistindo assim, na implementação do protocolo referido anteriormente.

### Interfaces

A execução do programa ocorre através de dois terminais distintos, um em cada computador, em que um dos computadores executa o programa em modo transmissor e o outro em modo recetor.

```
$ <PROGRAM> <SERIAL_PORT> <ROLE> <FILE>
   - PROGRAM: binário a executar
   - SERIAL_PORT: nome da porta série a utilizar
   - ROLE: 'tx' para o transmissor, 'rx' para o recetor
   - FILE: nome do ficheiro a enviar/receber
```

# Estrutura do Código

### *ApplicationLayer*

Uma vez que nesta camada não houve necessidade da criação de estruturas de dados auxiliares, apenas foram implementadas as seguintes funções:

```
void applicationLayer(const char *serialPort, const char *role, int baudRate, int
nTries, int timeout, const char *filename);
void auxRcvFileSize(unsigned char* packet, int size, unsigned long int *fileSize);
void auxDataPacket(const unsigned char* packet, const unsigned int packetSize,
unsigned char* buffer);
unsigned char * controlPacket(const unsigned int ctrlField, const char* filename,
long int length, unsigned int* size);
unsigned char * dataPacket(unsigned char seq, unsigned char *data, int dataSize,
int *packetSize);
```

### *LinkLayer*

Na implementação desta camada foram utilizadas três estruturas de dados auxiliares: *LinkLayer*, onde são caracterizados os parâmetros associados à transferência de dados, *LinkLayerRole*, que identifica o modo do computador em questão, isto é, se se trata de um transmissor ou de um recetor, *llState*, que identifica o estado da leitura e receção das tramas de informação.

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

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

```
typedef           enum
{
   START,
   FLAG_RCV,
   A_RCV,
   C_RCV,
   BCC1_CHECK,
   STOP_RCV,
   DATA_FOUND,
   READING_DATA,
   DISCONNECTED,
   BCC2_CHECK
} llState;
```

As funções implementadas foram:

```
// Open a connection using the "port" parameters defined in struct linkLayer.
int llopen(LinkLayer connectionParameters);
// Send data in buf with size bufSize.
int llwrite(const unsigned char *buf, int bufSize);
// Receive data in packet.
int llread(unsigned char *packet);
// Close previously opened connection.
// if showStatistics == TRUE, link layer should print statistics in the console on
close.
int llclose(int showStatistics);
unsigned char readSupervisionFrame();
int sendFrame(unsigned char A, unsigned char C);
```

# Casos de uso principais

O programa pode ser executado nos modos transmissor e recetor. Dependendo de qual for escolhido, as funções utilizadas e a sequência de chamadas serão diferentes.

### Transmissor

1. **llopen,** usada para estabelecer a ligação lógica entre o transmissor e o recetor, através da troca de pacotes de controlo e conexão com a porta série.
2. **controlPacket,** criação de um pacote de controlo.
3. **llwrite,** cria e envia pela porta série uma trama de informação com base no pacote recebido como argumento.
4. **readSupervisionFrame,** máquina de estados que lê e valida resposta do recetor após qualquer escrita na porta série.
5. **llclose,** usada para terminar a ligação entre o transmissor e o recetor, através da troca de pacotes de controlo.

### Recetor

1. **llread,** máquina de estados que gere e valida a receção de tramas de controlo e tramas de dados.

2. **sendFrame,** cria e envia pela porta série uma trama de supervisão com base na trama lida por llread().
3. **auxRcvFileSize,** retorna as características do ficheiro a ser transferido contidas no pacote de controlo em formato TLV passado como argumento.
4. **dataPacket,** retorna um segmento do ficheiro através do pacote de dados passado como argumento.

# Protocolo de ligação lógica

A camada responsável pela comunicação entre o emissor e o recetor é a camada de ligação de dados, que interage diretamente com a porta série. Para esse fim, utilizamos o protocolo *Stop-and-Wait* para o estabelecimento e terminação da ligação e para o envio de tramas de supervisão e informação.

A função **llopen** estabelece a ligação. Após a abertura e configuração da porta série, o emissor envia uma trama de supervisão SET para o recetor, ficando assim, à espera de receber uma trama de supervisão UA enviada pelo recetor, sendo que o recetor apenas responde com UA para sinalizar ao emissor que recebeu o SET. Assim sendo, é possível verificar se a ligação foi bem estabelecida se o emissor receber a trama UA. E após o estabelecimento da ligação ser verificado, é então possível para o emissor começar a enviar informação para posteriormente esta ser lida pelo recetor.

O envio de informação é feito pela função **llwrite**. Esta função recebe um pacote de controlo ou de dados e aplica-lhe a estratégia de *byte stuffing*, de modo a evitar conflitos com os bytes de dados que sejam iguais às *flags* da trama. Posteriormente transforma esse pacote numa trama de informação (*framing*), envia-se para o recetor e espera-se pela sua resposta. Se a trama for rejeitada, o envio é realizado novamente até ser aceite ou exceder o número máximo de tentativas de retransmissões. Cada tentativa de envio tem um limite de tempo após o qual ocorre time-out.

A função **llwrite** é a responsável pelo envio de informações. Para evitar conflitos com bytes de dados que sejam iguais às *flags* da trama, esta função recebe um pacote de dados ou de controlo e aplica a estratégia de *byte stuffing*. Posteriormente, esse pacote é transformado numa trama de informação (*framing*), sendo esta enviada ao recetor tendo-se, então, de esperar pela sua resposta. Se a trama for rejeitada, o envio é repetido até que seja aceite ou até que o número máximo de tentativas de retransmissão seja alcançado. Cada tentativa de envio tem uma duração máxima, após a qual ocorre uma interrupção (*time-out*).

A função **llread**, é então a responsável pela leitura de informação recebida pela porta série e verificação da sua validade. Começa então por fazer o *destuffing* do campo de dados de trama e valida o BCC1 e o BCC2, que por sua vez, verificam se ocorreu algum erro durante a transmissão da informação.

A função **llclose** é, por fim, a responsável pelo término da ligação, sendo invocada pelo emissor quando não houver mais retransmissões a realizar ou quando a transferência dos pacotes de dados foi dada como concluída. O emissor envia uma trama de supervisão DISC e espera que o recetor responda com a mesma trama. Quando o emissor recebe o DISC novamente, responde com a trama de supervisão UA e interrompe, por fim, a ligação.

# Protocolo de aplicação

A camada da aplicação é a que interage diretamente tanto com o ficheiro a ser transferido, tanto com o utilizador. É possível configurar, na mesma, qual o ficheiro a ser transferido, em que porta série, a velocidade da transferência, o número de bytes de dados do ficheiro inseridos em cada pacote, o número máximo de retransmissões e o tempo máximo de espera da resposta do recetor. A API da *LinkLayer*, que transforma pacotes de dados em tramas de informação, permite a transferência de ficheiros.

Posteriormente à ligação lógica entre o transmissor e o recetor ser estabelecida, todo o conteúdo do ficheiro é copiado para um buffer local e, em seguida, é fragmentado pela **applicationLayer** de acordo com o número de bytes especificado no argumento. O primeiro pacote enviado pelo transmissor contém dados no formato *TLV* (*Type*, *Length*, *Value*), criado pela função **controlPacket**, onde o tamanho do ficheiro é expresso em bytes. A função **auxControlPacket** descompacta esse pacote, no lado do recetor, para criar e alocar o espaço necessário para a receção do ficheiro.

A função **dataPacket** insere fragmentos do ficheiro a transferir num pacote de dados e a função **llwrite**, que se encontra na API, é usada para enviar o ficheiro pela porta série. Cada envio é acompanhado por uma resposta do recetor, indicando se aceita ou rejeita o pacote enviado. Caso aceite, o transmissor envia uma resposta com o fragmento seguinte, no entanto, caso rejeite, reenvia o mesmo fragmento. O recetor avalia cada pacote individualmente, através da função **llread**, e, por sua vez, a função **auxDataPacket** extrai do pacote o fragmento original do ficheiro quando recebido corretamente.

A ligação entre as duas máquinas é terminada quando a função **llclose** da API é chamada, após ocorrer o término da transferência de pacotes de dados, ou por não ser possível efetuar mais tentativas de retransmissão.

# Validação

Para testar a implementação do protocolo desenvolvido para este projeto, foram efetuados diversos testes:

- transferência de ficheiros de diferentes tamanhos
- transferência de ficheiros com nomes distintos
- transferência de ficheiros com diferentes *baudrates* (capacidades de ligação)
- transferência de pacotes de dados de diferentes tamanhos
- interrupção parcial e total da porta série
- introdução de ruído na porta série

Em todos os cenários apresentados o protocolo implementado garantiu a consistência do ficheiro transferido. Alguns dos testes foram também reproduzidos sob a presença do docente aquando da apresentação do projeto na aula laboratorial.

# Eficiência do protocolo de ligação de dados

## Variação do *baudrate* (capacidade de ligação)

Com um ficheiro de 10968 bytes, um tamanho da trama de 1000 bytes, a variação do *baudrate* origina os seguintes dados:

| Variação do *baudrate* | Tempo (s) | Received bitrate(bits/s) | Eficiência (%) |
|---|---|---|---|
| 9600 | 13,0026 | 6748 | 70,29 |
| 19200 | 7,43067 | 11808 | 61,5 |
| 38400 | 3,85608 | 22754 | 59,3 |
| 76800 | 2,14301 | 40944 | 53,3 |
| 115200 | 1,8304 | 47937 | 41,6 |

Como é possível observar, o *baudrate* e o tempo de transferência do ficheiro são inversamente proporcionais. A eficiência diminui com o aumento do *baudrate*, pois o aumento deste último não só aumento a velocidade de propagação como o número de erros.

## Variação do tamanho da trama

Com um ficheiro de 10968 bytes, um *baudrate* de 38400 bits/s, a variação do tamanho da trama origina os seguintes dados:

| Tamanho da Trama (bytes) | Tempo (s) | Received bitrate(bits/s) | Eficiência (%) |
|---|---|---|---|
| 200 | 4,501924 | 19490 | 50,7 |
| 600 | 4,119245 | 21300 | 55,47 |
| 1000 | 4,028288 | 21781 | 56,7 |
| 1500 | 4,003652 | 21916 | 57,07 |
| 2000 | 3,992041 | 21979 | 57,24 |

Como é possível observar, o tamanho da trama e o tempo de transferência do ficheiro são inversamente proporcionais. A eficiência aumenta com o aumento do tamanho da trama, pois é possível transmitir mais tramas em menos tempo. A acentuação da eficiência diminui à medida que se aumenta o tamanho da trama, uma vez que os números de tramas enviadas começam a ser muito semelhantes (depende do tamanho do ficheiro).

## Variação da taxa de erros

Com um ficheiro de 10968 bytes, um *baudrate* fixo de 38400 bits/s e tamanho da trama de 1500 bytes:

| FER (%) | Tempo (s) | Received bitrate(bits/s) | Eficiência (%) |
|---|---|---|---|
| 10 | 4,16903 | 21046 | 54,8 |
| 25 | 4,50274 | 19716 | 51,34 |
| 40 | 5,06201 | 17333 | 45,1 |
| 50 | 5,40862 | 16222 | 42,25 |
| 60 | 7,26468 | 12078 | 31,45 |

Como é possível observar, o FER e o tempo de transferência do ficheiro são diretamente proporcionais. Como visto no protocolo *Stop-And-Wait*, cada erro numa trama requere uma retransmissão, o que aumentará o tempo necessário para efetuar a transmissão de todas as tramas. Desta forma, podemos concluir que a FER é inversamente proporcional à eficiência.

# Conclusões

Concluindo, consideramos que o protocolo de ligação de dados teve uma enorme importância para a consolidação da matéria lecionada nas aulas teóricas, uma vez que a realização deste projeto nos possibilitou a interiorização dos conceitos de *byte stuffing*, *framing* e o funcionamento do protocolo *Stop-and-Wait* e como este deteta e lida com erros.

# Anexo I - 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;
}
```

# Anexo II - application_layer.h

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

#ifndef _APPLICATION_LAYER_H_
#define _APPLICATION_LAYER_H_

#include <stdio.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);
```

```
void auxRcvFileSize(unsigned char* packet, int size, unsigned long int *fileSize);

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

unsigned char * controlPacket(const unsigned int ctrlField, const char* filename,
long int length, unsigned int* size);

unsigned char * dataPacket(unsigned char seq, unsigned char *data, int dataSize,
int *packetSize);

#endif // _APPLICATION_LAYER_H_
```

# Anexo III - application_layer.c

```
// Application layer protocol implementation

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

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;

    int fd = llopen(connectionParameters);
    if (fd < 0) {
        perror("Connection error\n");
        exit(-1);
    }

    switch (connectionParameters.role) {

        case LlTx: {
```

```
            FILE* file = fopen(filename, "rb");
            if (file == NULL) {
                perror("File not found\n");
                exit(-1);
            }

            //Calculate file size
            int aux = ftell(file);
            fseek(file,0L,SEEK_END);
            long int fileSize = ftell(file) - aux;
            fseek(file, aux, SEEK_SET);

            //Write initial packet
            unsigned int ctrlPacketSize;
            unsigned char *initialPacket = controlPacket(2, filename, fileSize,
&ctrlPacketSize);
            if(llwrite(initialPacket, ctrlPacketSize) == -1){
                printf("Error in initial packet\n");
                exit(-1);
            }

            //Write data packets
            unsigned char seq = 0;
            unsigned char* content = (unsigned char*)malloc(sizeof(unsigned char) *
fileSize);
            fread(content, sizeof(unsigned char), fileSize, file);
            long int remainingBytes = fileSize;

            while (remainingBytes >= 0) {

                int dataSize = remainingBytes > (long int) MAX_PAYLOAD_SIZE ?
MAX_PAYLOAD_SIZE : remainingBytes;
                unsigned char* data = (unsigned char*) malloc(dataSize);
                memcpy(data, content, dataSize);
                int packetSize;
                unsigned char* packet = dataPacket(seq, data, dataSize,
&packetSize);

                if(llwrite(packet, packetSize) == -1) {
                    printf("Error in data packet\n");
                    exit(-1);
                }

                remainingBytes -= (long int) MAX_PAYLOAD_SIZE;
                content += dataSize;
                seq = (seq + 1) % 255;
            }

            //Write final packet
```

```
            unsigned char *finalPacket = controlPacket(3, filename, fileSize,
&ctrlPacketSize);
            if(llwrite(finalPacket, ctrlPacketSize) == -1) {
                printf("Error in final packet\n");
                exit(-1);
            }
            llclose(1);
            break;
        }

        case LlRx: {

            unsigned char *packet = (unsigned char *)malloc(MAX_PAYLOAD_SIZE);
            int packetSize = -1;

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

            //New file size
            unsigned long int rcvFSize = 0;
            auxRcvFileSize(packet, packetSize, &rcvFSize);

            //Write new file
            FILE* newFile = fopen((char *) filename, "wb+");

            while (1) {

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

                if(packetSize == 0) break; //Breaks if every packet has been read
                else if(packet[0] != 3){
                    unsigned char *buffer = (unsigned char*)malloc(packetSize);
                    auxDataPacket(packet, packetSize, buffer);
                    fwrite(buffer, sizeof(unsigned char), packetSize - 4, newFile);
                    free(buffer);
                }
                else continue;
            }

            fclose(newFile);
            break;
        }
        default:
            exit(-1);
            break;
    }
}

void auxRcvFileSize(unsigned char* packet, int size, unsigned long int *fileSize) {

    // File Size
```

```
    unsigned char fSizeB = packet[2];
    unsigned char fSizeAux[fSizeB];
    memcpy(fSizeAux, packet + 3, fSizeB);
    for(unsigned int i = 0; i < fSizeB; i++)
        *fileSize |= (fSizeAux[fSizeB-i-1] << (8*i));

}

unsigned char * controlPacket(const unsigned int ctrlField, const char* filename,
long int length, unsigned int* size){

    int len1 = 0;
    unsigned int tmp = length;
    while (tmp > 1) {
        tmp >>= 1;
        len1++;
    }
    len1 = (len1 + 7) / 8; //file size (bytes)
    const int len2 = strlen(filename); //file name (bytes)
    *size = 5 + len1 + len2;
    unsigned char *packet = (unsigned char*)malloc(*size);

    unsigned int pos = 0;
    packet[pos++] = ctrlField;
    packet[pos++] = 0; // T_1 (0 = file size)
    packet[pos++] = len1; // L_1

    for (unsigned char i = 0 ; i < len1 ; i++) {
        packet[2+len1-i] = length & 0xFF;
        length >>= 8; // V_1
    }

    pos += len1;
    packet[pos++] = 1; // T_2 (1 = file name)
    packet[pos++] = len2; // L_2

    memcpy(packet + pos, filename, len2); // V_2

    return packet;
}

unsigned char * dataPacket(unsigned char seq, unsigned char *data, int dataSize,
int *packetSize){

    *packetSize = 4 + dataSize;
    unsigned char* packet = (unsigned char*)malloc(*packetSize);

    packet[0] = 1;
    packet[1] = seq;
    packet[2] = dataSize >> 8 & 0xFF;
```

```
    packet[3] = dataSize & 0xFF;
    memcpy(packet + 4, data, dataSize);

    return packet;
}

void auxDataPacket(const unsigned char* packet, const unsigned int packetSize,
unsigned char* buffer) {
    memcpy(buffer, packet + 4, packetSize - 4);
    buffer += (packetSize + 4);
}
```

# Anexo IV - 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;

typedef enum
{
   START,
   FLAG_RCV,
   A_RCV,
   C_RCV,
   BCC1_CHECK,
   STOP_RCV,
   DATA_FOUND,
   READING_DATA,
   DISCONNECTED,
   BCC2_CHECK
} llState;
```

```c
// 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

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

#define _POSIX_SOURCE 1 // POSIX compliant source
#define BAUDRATE 9600

#define BUF_SIZE 256

#define FLAG 0x7E
#define ESC 0x7D
#define A_TX 0x03
#define A_RX 0x01
#define C_SET 0x03
#define C_DISC 0x0B
#define C_UA 0x07
#define C_RR(Nr) ((Nr << 7) | 0x05)
#define C_REJ(Nr) ((Nr << 7) | 0x01)
#define C_N(Ns) (Ns << 6)

// 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.
```

```c
// Return "1" on success or "-1" on error.
int llclose(int showStatistics);

unsigned char readSupervisionFrame();

int sendFrame(unsigned char A, unsigned char C);

#endif // _LINK_LAYER_H_
```

# Anexo V - link_layer.c

```c
// Link layer protocol implementation

#include "link_layer.h"

volatile int STOP = FALSE;
int fd = 0;
int alarmEnabled = FALSE;
int alarmCount = 0;
int timeout = 0;
int retransmissions = 0;
unsigned char tramaTx = 0;
unsigned char tramaRx = 1;
clock_t start_time;

void alarmHandler(int signal) {
    alarmEnabled = TRUE;
    alarmCount++;
       printf("Alarm #%d\n", alarmCount);
}

int connection(const char *serialPort) {

    int fd = open(serialPort, O_RDWR | O_NOCTTY);
    if (fd < 0) {
        perror(serialPort);
        return -1;
    }

    struct termios oldtio;
    struct termios newtio;

    if (tcgetattr(fd, &oldtio) == -1)
    {
        perror("tcgetattr");
        exit(-1);
    }

    memset(&newtio, 0, sizeof(newtio));
```

```
    newtio.c_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;
    newtio.c_iflag = IGNPAR;
    newtio.c_oflag = 0;
    newtio.c_lflag = 0;
    newtio.c_cc[VTIME] = 0;
    newtio.c_cc[VMIN] = 0;

    tcflush(fd, TCIOFLUSH);

    if (tcsetattr(fd, TCSANOW, &newtio) == -1) {
        perror("tcsetattr");
        return -1;
    }

    return fd;
}

////////////////////////////////////////////////
// LLOPEN
////////////////////////////////////////////////
int llopen(LinkLayer connectionParameters) {

    llState state = START;
    fd = connection(connectionParameters.serialPort);
    if (fd < 0) return -1;

    unsigned char byte;
    timeout = connectionParameters.timeout;
    retransmissions = connectionParameters.nRetransmissions;
    switch (connectionParameters.role) {

        case LlTx: {
                    start_time = clock();
            (void) signal(SIGALRM, alarmHandler);
            while (connectionParameters.nRetransmissions != 0 && state != STOP_RCV)
{

                if(sendFrame(A_TX, C_SET) < 0){printf("Send Frame Error\n"); return
-1;}
                alarm(connectionParameters.timeout);
                alarmEnabled = FALSE;

                while (alarmEnabled == FALSE && state != STOP_RCV) {
                    if (read(fd, &byte, 1) > 0) {
                        switch (state) {
                            case START:
                                if (byte == FLAG) state = FLAG_RCV;
                                break;
                            case FLAG_RCV:
```

```
                            if (byte == A_RX) state = A_RCV;
                            else if (byte != FLAG) state = START;
                            break;
                        case A_RCV:
                            if (byte == C_UA) state = C_RCV;
                            else if (byte == FLAG) state = FLAG_RCV;
                            else state = START;
                            break;
                        case C_RCV:
                            if (byte == (A_RX ^ C_UA)) state = BCC1_CHECK;
                            else if (byte == FLAG) state = FLAG_RCV;
                            else state = START;
                            break;
                        case BCC1_CHECK:
                            if (byte == FLAG) state = STOP_RCV;
                            else state = START;
                            break;
                        default:
                            break;
                    }
                }
            }
            connectionParameters.nRetransmissions--;
        }
        if (state != STOP_RCV) return -1;
        break;
    }

    case LlRx: {

        while (state != STOP_RCV) {
            if (read(fd, &byte, 1) > 0) {
                switch (state) {
                    case START:
                        if (byte == FLAG) state = FLAG_RCV;
                        break;
                    case FLAG_RCV:
                        if (byte == A_TX) state = A_RCV;
                        else if (byte != FLAG) state = START;
                        break;
                    case A_RCV:
                        if (byte == C_SET) state = C_RCV;
                        else if (byte == FLAG) state = FLAG_RCV;
                        else state = START;
                        break;
                    case C_RCV:
                        if (byte == (A_TX ^ C_SET)) state = BCC1_CHECK;
                        else if (byte == FLAG) state = FLAG_RCV;
                        else state = START;
                        break;
```

```
                        case BCC1_CHECK:
                            if (byte == FLAG) state = STOP_RCV;
                            else state = START;
                            break;
                        default:
                            break;
                    }
                }
            }
            if(sendFrame(A_RX, C_UA) < 0){printf("Send Frame Error\n"); return -1;}
            break;
        }
        default:
            return -1;
            break;
    }
    return fd;
}

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

    int frameSize = 6 + bufSize;
    unsigned char *frame = (unsigned char *) malloc(frameSize);
    frame[0] = FLAG;
    frame[1] = A_TX;
    frame[2] = C_N(tramaTx);
    frame[3] = (frame[1] ^ frame[2]);
    memcpy(frame + 4, buf, bufSize);

    unsigned char BCC2 = buf[0];
    for (unsigned int i = 1 ; i < bufSize ; i++) BCC2 ^= buf[i];

    //Byte stuffing
    int j = 4;
    for (unsigned int i = 0 ; i < bufSize ; i++) {
        if(buf[i] == FLAG || buf[i] == ESC) {
            frame = realloc(frame,++frameSize);
            frame[j++] = ESC;
        }
        frame[j++] = buf[i];
    }
    frame[j++] = BCC2;
    frame[j++] = FLAG;

    int currentTransmition = 0;
    int rejected = 0, accepted = 0;
```

```
    while (currentTransmition < retransmissions) {
        alarmEnabled = FALSE;
        alarm(timeout);
        rejected = 0;
        accepted = 0;
        while (alarmEnabled == FALSE && !rejected && !accepted) {

            if(write(fd, frame, j) < 0) return -1;
            unsigned char controlField = readSupervisionFrame();

            if(!controlField){
                continue;
            }
            else if(controlField == C_REJ(0) || controlField == C_REJ(1)) {
                rejected = 1;
            }
            else if(controlField == C_RR(0) || controlField == C_RR(1)) {
                accepted = 1;
                tramaTx = (tramaTx + 1) % 2;  //Nr module-2 counter (enables to
distinguish frame 0 and frame 1)
            }
            else continue;

        }
        if (accepted) break;
        currentTransmition++;
    }

    free(frame);
    if(accepted) return frameSize;
    else{
        llclose(1);
        return -1;
    }
}

////////////////////////////////////////////////
// LLREAD
////////////////////////////////////////////////
int llread(unsigned char *packet) {

    unsigned char byte, ctrlField;
    int i = 0;
    llState state = START;

    while (state != STOP_RCV) {
        if (read(fd, &byte, 1) > 0) {
            switch (state) {
                case START:
                    if (byte == FLAG) state = FLAG_RCV;
```

```
                    break;
                case FLAG_RCV:
                    if (byte == A_TX) state = A_RCV;
                    else if (byte != FLAG) state = START;
                    break;
                case A_RCV:
                    if (byte == C_N(0) || byte == C_N(1)){
                        state = C_RCV;
                        ctrlField = byte;
                    }
                    else if (byte == FLAG) state = FLAG_RCV;
                    else if (byte == C_DISC) {
                        if(sendFrame(A_RX, C_DISC) < 0){printf("Send Frame
Error\n"); return -1;}
                        return 0;
                    }
                    else state = START;
                    break;
                case C_RCV:
                    if (byte == (A_TX ^ ctrlField)) state = READING_DATA;
                    else if (byte == FLAG) state = FLAG_RCV;
                    else state = START;
                    break;
                case READING_DATA:
                    if (byte == ESC) state = DATA_FOUND;
                    else if (byte == FLAG){
                        unsigned char bcc2 = packet[i-1];
                        i--;
                        packet[i] = '\0';
                        unsigned char bcc2Check = packet[0];

                        for (unsigned int j = 1; j < i; j++)
                            bcc2Check ^= packet[j];

                        if (bcc2 == bcc2Check){
                            state = STOP_RCV;
                            if(sendFrame(A_RX, C_RR(tramaRx)) < 0){printf("Send
Frame Error\n");}
                            tramaRx = (tramaRx + 1) % 2; //Ns module-2 counter
(enables to distinguish frame 0 and frame 1)
                            return i;
                        }
                        else{
                            printf("Retransmission Error\n");
                            sendFrame(A_RX, C_REJ(tramaRx));
                            return -1;
                        };

                    }
                    else{
```

```
                        packet[i++] = byte;
                    }
                    break;
                case DATA_FOUND:
                    state = READING_DATA;
                    if (byte == ESC || byte == FLAG) packet[i++] = byte;
                    else{
                        packet[i++] = ESC;
                        packet[i++] = byte;
                    }
                    break;
                default:
                    break;
            }
        }
    }
    return -1;
}

////////////////////////////////////////////////
// LLCLOSE
////////////////////////////////////////////////
int llclose(int showStatistics) {

    llState state = START;
    unsigned char byte;
    (void) signal(SIGALRM, alarmHandler);

    while (retransmissions != 0 && state != STOP_RCV) {

        if(sendFrame(A_TX, C_DISC) < 0){printf("Send Frame Error\n"); return -1;}
        alarm(timeout);
        alarmEnabled = FALSE;

        while (alarmEnabled == FALSE && state != STOP_RCV) {
            if (read(fd, &byte, 1) > 0) {
                switch (state) {
                    case START:
                        if (byte == FLAG) state = FLAG_RCV;
                        break;
                    case FLAG_RCV:
                        if (byte == A_RX) state = A_RCV;
                        else if (byte != FLAG) state = START;
                        break;
                    case A_RCV:
                        if (byte == C_DISC) state = C_RCV;
                        else if (byte == FLAG) state = FLAG_RCV;
                        else state = START;
                        break;
                    case C_RCV:
```

```
                        if (byte == (A_RX ^ C_DISC)) state = BCC1_CHECK;
                        else if (byte == FLAG) state = FLAG_RCV;
                        else state = START;
                        break;
                    case BCC1_CHECK:
                        if (byte == FLAG) state = STOP_RCV;
                        else state = START;
                        break;
                    default:
                        break;
                }
            }
        }
        retransmissions--;
    }

    if (state != STOP_RCV) return -1;
    if(sendFrame(A_TX, C_UA) < 0) return -1;
      if (showStatistics == 1) {
            clock_t end_time = clock();

            double elapsed_time = (double)(end_time - start_time) /
CLOCKS_PER_SEC;
            printf("Elapsed time: %f seconds\n", elapsed_time);
      }
    return close(fd);
}

unsigned char readSupervisionFrame() {

    unsigned char byte, ctrlField = 0;
    llState state = START;

    while (state != STOP_RCV && alarmEnabled == FALSE) {
        if (read(fd, &byte, 1) > 0 || 1) {
            switch (state) {
                case START:
                    if (byte == FLAG) state = FLAG_RCV;
                    break;
                case FLAG_RCV:
                    if (byte == A_RX) state = A_RCV;
                    else if (byte != FLAG) state = START;
                    break;
                case A_RCV:
                    if (byte == C_RR(0) || byte == C_RR(1) || byte == C_REJ(0) ||
byte == C_REJ(1) || byte == C_DISC){
                        state = C_RCV;
                        ctrlField = byte;
                    }
                    else if (byte == FLAG) state = FLAG_RCV;
```

```
                    else state = START;
                    break;
                case C_RCV:
                    if (byte == (A_RX ^ ctrlField)) state = BCC1_CHECK;
                    else if (byte == FLAG) state = FLAG_RCV;
                    else state = START;
                    break;
                case BCC1_CHECK:
                    if (byte == FLAG){
                        state = STOP_RCV;
                    }
                    else state = START;
                    break;
                default:
                    break;
            }
        }
    }
    return ctrlField;
}

// Send Supervision Frame
int sendFrame(unsigned char A, unsigned char C) {
    unsigned char buffer[5] = {FLAG, A, C, A ^ C, FLAG};
    return write(fd, buffer, 5);
}
```