



# Protocolo de Ligação de Dados

1º Trabalho Laboratorial

Redes de Computadores

Grupo 1 – Turma 8

Licenciatura em Engenharia Informática e Computação

Domingos José Silva Moreira dos Santos up201906680

Jorge Levi Perdigoto da Costa up201706518

Vitor de Sá Carneiro Bizarro up202007888

# Índice

Sumário	3
1. Introdução	3
2. Arquitetura	4
3. Estrutura do código	4
4. Casos de uso principais	6
5. Protocolo de ligação lógica	6
6. Protocolo de aplicação	7
7. Validação	8
8. Eficiência do protocolo de ligação de dados	10
9. Conclusões	11
Anexos	11

# Sumário

Este presente relatório tem como fim destrinchar o desenvolvimento do 1º trabalho laboratorial da disciplina de Redes de Computadores, que consistia na transferência de dados entre 2 computadores conectados por portas séries. A demonstração em questão se deu pela partição em tramas de informação de uma imagem.

O projeto foi demonstrado com sucesso, tendo a imagem sendo transferida integralmente de uma máquina à outra de modo satisfatório. Entretanto, quando perdida a ligação o recetor não envia a resposta REJ corretamente, sendo perdidos os dados e consequentemente, o ficheiro não é transferido corretamente. Posteriormente foi ofertada a possibilidade de corrigir o erro, tendo a equipa estado satisfeita com o trabalho realizado e escolhido não voltar a alterar o programa.

## 1. Introdução

Neste trabalho, foram aplicados os conhecimentos adquiridos ao longo de metade do semestre. Dentre estes conhecimentos estão a leitura e escrita de ficheiros, uso de máquinas de estado, alarmes, cabos séries, transferência de dados a baixo nível. Neste relatório encontra-se discriminado toda a linha de pensamento por trás do programa, dividida nas seguintes secções:

### 1. Introdução

*Indicação dos objetivos do trabalho e do relatório e descrição da lógica do relatório com indicações sobre o tipo de informação que poderá ser encontrada em cada uma secções seguintes.*

### 2. Arquitetura

*Explicação de como a aplicação está planeada em alto nível organizacional. Divisões em interfaces e funções.*

### 3. Estrutura do código

*Como as estruturas de dados funcionam e se relacionam concretamente com a arquitetura.*

### 4. Casos de uso principais

*Identificação e detalhamento das funções orgânicas.*

5. **Protocolo de ligação lógica**

*Como as funções orgânicas foram implementadas em código no contexto das ligações inter computadores.*

6. **Protocolo de aplicação**

*Como as funções orgânicas foram implementadas em código no contexto da aplicação.*

7. **Validação**

*Descrição dos testes efetuados com apresentação dos resultados.*

8. **Eficiência do protocolo de ligação de dados**

*Caracterização estatística da eficiência do protocolo, efetuada recorrendo a medidas sobre o código desenvolvido.*

9. **Conclusões**

*Síntese da informação apresentada nas seções anteriores; reflexão sobre os objetivos de aprendizagem alcançados.*

Além destas subdivisões, encontra-se a o fim deste relatório os anexos do código-fonte fundamentais.

## 2. Arquitetura

A arquitetura do programa não divergiu da proposta feita pelos professores da unidade curricular. Conforme o código base que nos foi disponibilizado, a arquitetura se orienta a partir de dois principais ficheiros: o **application\_layer.c** e o **link\_layer.c**

O **application\_layer.c**, por sua vez, serve como ponte entre utilizador e camada de ligação. É o artefato inicializador do programa e através dele é determinado todo o funcionamento do programa, desde a baudrate até as portas séries. Esta camada também determina o funcionamento do transmissor e do receptor a alternância entre eles.

Já o **link\_layer.c**, é onde estão implementadas as principais funções invocadas na aplicação (**llopen**, **llread**, **llwrite**, **llclose**), que serão detalhadas a seguir. Para além destas funções principais é onde estão implementadas as funções secundárias, como as do alarme e a da máquina de estados, a serem utilizadas na leitura da imagem.

## 3. Estrutura do código

O código está estruturado em três ficheiros: **main.c**, **application\_layer.c** e **link\_layer.c**.

## **main.c**

- Recebe os argumentos da linha de comandos para a execução do programa.

## **application\_layer.c**

### **Funções:**

#### **applicationLayer:**

- Armazena os argumentos do main numa estrutura **LinkLayer**.
- Executa llopen, chama função transmitter ou receiver dependendo dos args, fecha com llclose.

#### **transmitter:**

- Abre o ficheiro, divide em pacotes e executa llwrite de cada pacote.

#### **receiver:**

- Recebe os pacotes com llread, faz parse dos mesmos e escreve no ficheiro de destino.

## **link\_layer.c**

### **Funções:**

#### **llopen:**

- Abre a porta série e troca tramas SET e UA.

#### **llwrite:**

- Envia tramas I.

#### **llread:**

- Recebe tramas I.

#### **llclose:**

- Troca tramas DISC e UA e fecha a ligação à porta série.

#### **calculateBCC2:**

- Calcula BCC2.

#### **stuffing, destuffing:**

- Efetua stuffing e destuffing da trama.

#### **alarmHandler:**

- Contador do alarme e ativa flag.

#### **sendBuffer:**

- Preenche e envia trama de Controlo.

#### **stateMachine:**

- Lê um byte da porta série e muda ou não o estado da máquina de estados.

## 4. Casos de uso principais

- O trabalho consiste em dois principais casos de uso: a transmissão de um ficheiro de um computador para outro, e um método e estrutura que permitem ao transmissor escolher o ficheiro / dados para transmitir, via porta série.
- Relativamente ao método, o utilizador deverá inserir um conjunto de argumentos, de modo a iniciar a aplicação. Deste modo, o emissor deve escolher a porta série adequada (ex: /dev/ttyS0), bem como o ficheiro a ser enviado (ex: pinguim.gif).
- Como tal, a transmissão de dados ocorre na seguinte sequência:
  - O emissor escolhe o ficheiro a ser enviado;
  - Procede-se à configuração da ligação entre os dois dispositivos;
  - Estabelecimento da ligação entre eles;
  - Emissor/Transmissor emite os dados;
  - Recetor recebe os mesmos;
  - Recetor guarda os dados num ficheiro com o mesmo nome que é recebido;
  - Termina a ligação/transfêrencia;

## 5. Protocolo de ligação lógica

- LLOPEN
  - De modo a estabelecer uma conexão entre o emissor e o receptor, foi criada esta função.
  - O emissor envia a trama SET (Controlo) e ativa o time-counter, desativado depois de receber a resposta UA;
  - Caso não receba um UA dentro de um determinado timer-counter, o SET é reenviado até atingir um limite de “recalls”. Caso não obtiver o UA nesse limite, o programa termina;
  - Por outro lado, o recetor, espera um SET para enviar um UA;
- LLWRITE
  - Esta função do emissor permite o envio de tramas e o stuffing das mesmas;
  - Inicialmente, ocorre o framing da mensagem a ser enviada, ou seja, cria-se o cabeçalho do Protocolo de Ligação (recorrendo ao BCC2);
  - O envio da trama tem o mesmo sistema de timeout e retransmissão até obter sucesso, nomeadamente no envio do SET no LLOPEN;
  - Como tal, se for recebido um REJ, a mensagem é reenviada.
- LLREAD
  - Esta função permite ao recetor receber a mensagem enviada pelo emissor, caracter a caracter, verificando o BCC2;

- Caso a leitura esteja correta, é enviado um RR, caso contrário, um REJ;
- De seguida, é feito o destuffing da trama;
- LLCLOSE
  - De modo a terminar uma conexão entre o emissor e o receptor, foi criada esta função.
  - O emissor envia a trama *DISC* (Supervisão) e termina com o envio de uma UA;
  - Por outro lado, no recetor, é enviado um DISC e esperado um UA;

## 6. Protocolo de aplicação

O protocolo de aplicação, representado no ficheiro **application\_layer.c** serve como ponte entre o utilizador e a camada de ligação. É o artefato inicializador do programa e através dele é determinado todo o funcionamento do programa. Esta camada também determina o funcionamento do transmissor e do receptor a alternância entre eles.

Para o pleno funcionamento da camada de aplicação foi introduzida uma estrutura que armazena desde a baudrate até as portas séries. Foi definida na header **link\_layer.h**

```

23
24  typedef struct
25  {
26      char serialPort[50];
27      LinkLayerRole role;
28      int baudRate;
29      int nRetransmissions;
30      int timeout;
31  } LinkLayer;
32

```

Através do atributo “role” é sabido que tipo de ação é tomada, em que “tx” configura o caso do transmissor enquanto “rx” o receptor.

No caso do transmissor, o ficheiro em questão é aberto e as informações passam a ser enviadas em packets em consonância com o tamanho pré-definido. Através de funções **llwrite**, discriminada logo acima, são enviados packets de 3 tipos, um inicial, um ou vários intermediários e um final. Depois de enviado por completo, o ficheiro é fechado.

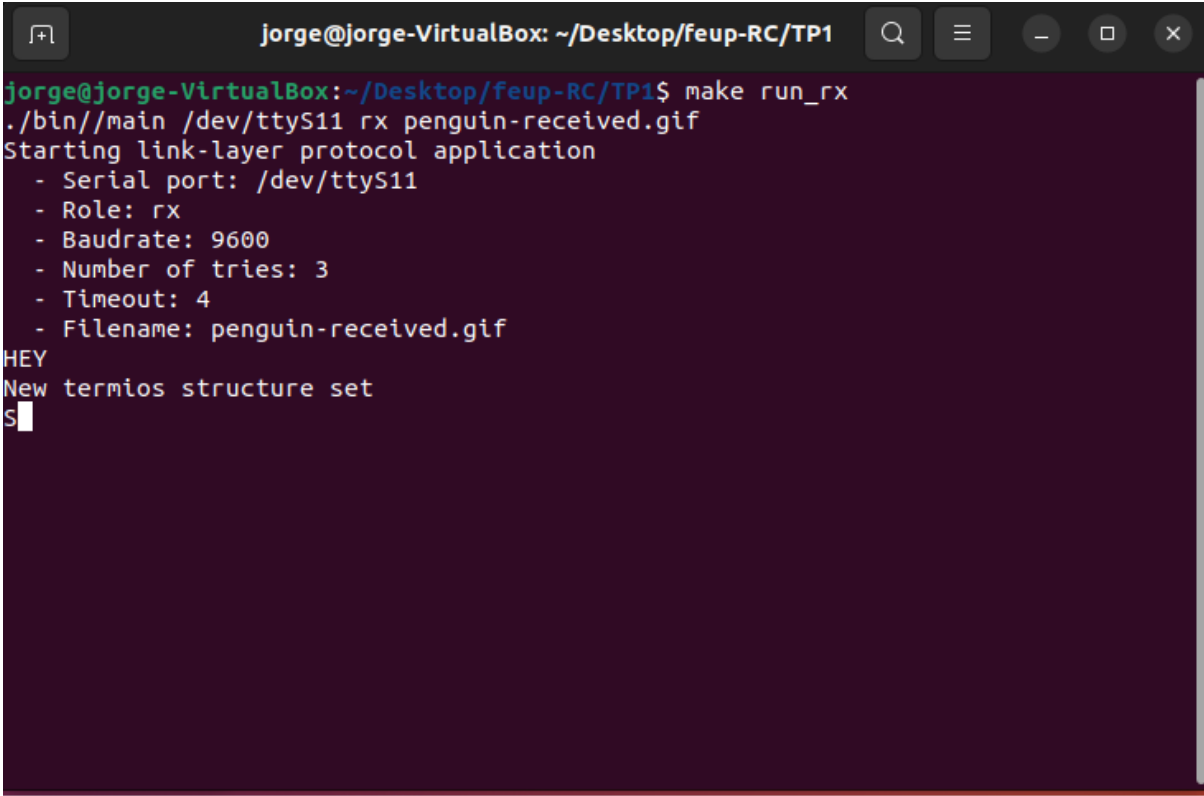
No caso do receptor, o ciclo fica à espera da recepção e leitura, através da **llread**, de todos os packets do ficheiro para que o programa possa seguir, de modo similar ao que acontece no alarme, sendo que sem tempo definido. O ficheiro também é fechado.

Para todos os casos há preparo de resguardo em caso de erro.

## 7. Validação

Os terminais representados pelo **receptor** e pelo **transmissor** estão representados nos screenshots a seguir ao longo do funcionamento da aplicação.

**Receptor** ao começo:



```
jorge@jorge-VirtualBox: ~/Desktop/feup-RC/TP1
jorge@jorge-VirtualBox:~/Desktop/feup-RC/TP1$ make run_rx
./bin//main /dev/ttyS11 rx penguin-received.gif
Starting link-layer protocol application
- Serial port: /dev/ttyS11
- Role: rx
- Baudrate: 9600
- Number of tries: 3
- Timeout: 4
- Filename: penguin-received.gif
HEY
New termios structure set
S
```

**Transmissor** ao começo:



```
jorge@jorge-VirtualBox: ~/Desktop/feup-RC/TP1
- Number of tries: 3
- Timeout: 4
- Filename: penguin.gif
HEY
tcsetattr: Invalid argument
make: *** [Makefile:31: run_tx] Erro 255
jorge@jorge-VirtualBox:~/Desktop/feup-RC/TP1$ make run_tx
./bin//main /dev/ttyS10 tx penguin.gif
Starting link-layer protocol application
- Serial port: /dev/ttyS10
- Role: tx
- Baudrate: 9600
- Number of tries: 3
- Timeout: 4
- Filename: penguin.gif
HEY
New termios structure set
5 bytes written
```

**Receptor** ao final:

```
jorge@jorge-VirtualBox: ~/Desktop/feup-RC/TP1
STATUS 1 ALL OK: db , 40
SENDING RESPONSE
PACKET : 3

ENDING PACKET
START
FLAG_RCV
A_RCV
C_RCV
-1
BCC_OK
STOP
Received DISC
Sent DISC
5 bytes written
START
FLAG_RCV
A_RCV
C_RCV
-1
BCC_OK
STOP
Received UA
jorge@jorge-VirtualBox:~/Desktop/feup-RC/TP1$
```

**Transmissor** ao final:

```
jorge@jorge-VirtualBox: ~/Desktop/feup-RC/TP1
END STUFFING
Data Enviada. 30 bytes written, 1º try...
START
FLAG_RCV
A_RCV
C_RCV
0
BCC_OK
STOP
Data Accepted!
Ending packet sent
Sent DISC
5 bytes written
START
FLAG_RCV
A_RCV
C_RCV
-1
BCC_OK
STOP
Received DISC
Sent UA
5 bytes written
jorge@jorge-VirtualBox:~/Desktop/feup-RC/TP1$
```

## 8. Eficiência do protocolo de ligação de dados

O programa foi testado com mais ficheiros de tamanhos variáveis, sendo todos transferidos corretamente, e com um clock e argumento st\_size de stat, verificámos o tempo e tamanho da transferência;

- |                    | Tempo       | Tamanho       | Velocidade   |
|--------------------|-------------|---------------|--------------|
| • <i>Penguin:</i>  | 13.94 sec,  | 10968 bytes,  | 786.487 B/s  |
| • <i>Imagem 2:</i> | 57.45 sec,  | 45421 bytes,  | 790.559 B/s  |
| • <i>Imagem 3:</i> | 130.46 sec, | 103971 bytes, | 796.9218 B/s |
| • <i>Imagem 4:</i> | 38.508 sec, | 30484 bytes,  | 791.6137 B/s |
| • <i>Imagem 5:</i> | 92.308 sec, | 73173 bytes,  | 792.704 B/s  |

## 9. Conclusões

Este projecto laboratorial permitiu-nos colocar em prática alguns dos conhecimentos que só tínhamos tido a oportunidade de discutir de modo teórico. Foram aprofundadas técnicas como a gestão e organização de camadas e interfaces, e sobretudo como estas podem ser usadas a nível da aplicação e ligação.

O relatório também nos permitiu fazer a ponte entre a esfera prática e teórica, de modo que conseguimos perceber melhor como ambas funcionam.

Conclui-se que o projeto correu com sucesso, uma vez que erros previstos foram suportados e a imagem foi bem partilhada.

## Anexos

### application\_layer.c:

```
// Application layer protocol implementation

#include "application_layer.h"

int transmitter(const char *filename)
{
    struct stat file_stat;
    if (stat(filename, &file_stat) < 0)
    {
```

```

        perror("Error getting file information.");
        return -1;
    }

    FILE *fptr = fopen(filename, "rb");

    // Starting packet
    unsigned L1 = sizeof(file_stat.st_size);
    unsigned L2 = strlen(filename);
    unsigned packet_size = 5 + L1 + L2;

    unsigned char packet[packet_size];
    packet[0] = STARTING_PACKET;
    packet[1] = FILE_SIZE;
    packet[2] = L1;
    memcpy(&packet[3], &file_stat.st_size, L1);
    packet[3 + L1] = FILE_NAME;
    packet[4 + L1] = L2;
    memcpy(&packet[5 + L1], filename, L2);

    if (llwrite(packet, packet_size) < 0)
        return -1;

    printf("Starting packet sent\n");

    // Middle packets
    unsigned char buf[MAX_PACKET_SIZE];
    unsigned bytes_to_send;
    unsigned sequenceNumber = 0;

    while ((bytes_to_send = fread(buf, sizeof(unsigned char),
MAX_PACKET_SIZE - 4, fptr)) > 0)
    {
        printf("MIDDLE PACKET\n");
        unsigned char dataPacket[MAX_PACKET_SIZE];
        dataPacket[0] = MIDDLE_PACKET;
        dataPacket[1] = sequenceNumber % 255;
        dataPacket[2] = (bytes_to_send / 256);
        dataPacket[3] = (bytes_to_send % 256);
        memcpy(&dataPacket[4], buf, bytes_to_send);

        llwrite(dataPacket, ((bytes_to_send + 4) < MAX_PACKET_SIZE) ?
(bytes_to_send + 4) : MAX_PACKET_SIZE);
    }

```

```

        printf("Sent %d° data package\n", sequenceNumber);
        sequenceNumber++;
    }

    printf("Midle packets sent\n");

    // Ending packet
    L1 = sizeof(file_stat.st_size);
    L2 = strlen(filename);
    packet_size = 5 + L1 + L2;

    packet[packet_size];
    packet[0] = ENDING_PACKET;
    packet[1] = FILE_SIZE;
    packet[2] = L1;
    memcpy(&packet[3], &file_stat.st_size, L1);
    packet[3 + L1] = FILE_NAME;
    packet[4 + L1] = L2;
    memcpy(&packet[5 + L1], filename, L2);

    if (llwrite(packet, packet_size) < 0)
        return -1;

    printf("Ending packet sent\n");

    fclose(fptr);

    return 0;
}

int receiver(const char *filename)
{
    int s;
    int nump = 0;
    int numTries = 0;
    static FILE *destination;
    int stop = FALSE;
    while (stop == FALSE)
    {
        printf("\nRECEIVER\n");

        unsigned char buf[MAX_PACKET_SIZE];
        if ((s = llread(buf)) < 0)

```

```

        continue;
    printf("PACKET : %d\n", buf[0]);
    switch (buf[0])
    {
    case STARTING_PACKET:
        destination = fopen(filename, "wb");
        break;

    case MIDDLE_PACKET:
        static unsigned int n = 0;
        if (buf[1] != n)
            return -1;
        n = (n + 1) % 255;

        unsigned int data_size = buf[2] * 256 + buf[3];
        fwrite(&buf[4], sizeof(unsigned char), data_size *
sizeof(unsigned char), destination);

        break;

    case ENDING_PACKET:
        printf("\nENDING PACKET\n");
        close(destination);
        stop = TRUE;
        break;
    }
}
return 0;
}

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

    layer.baudRate = baudRate;
    layer.nRetransmissions = nTries;

    if (strcmp(role, "tx") == 0)
        layer.role = LlTx;
    if (strcmp(role, "rx") == 0)
        layer.role = LlRx;
}

```

```

    sprintf(layer.serialPort, "%s", serialPort);
    layer.timeout = timeout;

    if (llopen(layer) < 0)
        return -1;

    switch (layer.role)
    {
    case LlTx:
        transmitter(filename);
        break;
    case LlRx:
        receiver(filename);
        break;
    default:
        break;
    }

    llclose(FALSE);
}

```

### link\_layer.c:

// Link layer protocol implementation

```

#include "link_layer.h"

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

struct termios oldtio;
struct termios newtio;

int alarmEnabled = FALSE;
int alarmCount = 0;

volatile int STOP = FALSE;

```

```

int state = START;

int fd;

int bytes;

unsigned char readbyte;

int response = -1;

LinkLayer layer;

LinkLayerRole getRole()
{
    return layer.role;
}

int getnTransmissions()
{
    return layer.nRetransmissions;
}

int getTimeOut()
{
    return layer.timeout;
}

int stateMachine(unsigned char a, unsigned char c, int isData, int
RR_REJ)
{
    if (!RR_REJ)
        response = -1;
    unsigned char byte = 0;

    int bytes = 0;

    bytes = read(fd, &byte, 1);

    if (bytes > 0)
    {
        switch (state)
        {

```



```

case START:
    printf("START\n");
    if (byte == FLAG)
        state = FLAG_RCV;
    break;
case FLAG_RCV:
    printf("FLAG_RCV\n");
    if (byte == a)
        state = A_RCV;
    else if (byte != FLAG)
        state = START;
    break;
case A_RCV:
    printf("A_RCV\n");
    if (RR_REJ)
    {
        switch (byte)
        {
            case RR(0):
                response = RR0;
                state = C_RCV;
                break;
            case RR(1):
                response = RR1;
                state = C_RCV;
                break;
            case REJ(0):
                response = REJ0;
                state = C_RCV;
                break;
            case REJ(1):
                response = REJ1;
                state = C_RCV;
                break;
            default:
                state = START;
                break;
        }
    }
    else
    {
        if (byte == c)
            state = C_RCV;
    }
}

```

```

        else if (byte == FLAG)
            state = FLAG_RCV;
        else
            state = START;
    }
    break;
case C_RCV:
    printf("C_RCV\n");
    switch (response)
    {
    case RR0:
        c = RR(0);
        break;
    case RR1:
        c = RR(1);
        break;
    case REJ0:
        c = REJ(0);
        break;
    case REJ1:
        c = REJ(1);
        break;
    default:
        break;
    }
    printf("%d\n", response);
    if (byte == (a ^ c))
    {
        if (isData)
            state = WAITING_DATA;
        else
            state = BCC_OK;
    }
    else if (byte == FLAG)
        state = FLAG_RCV;
    else
        state = START;
    break;
case WAITING_DATA:
    printf("WAITING_DATA\n");
    if (byte == FLAG)
    {
        STOP = TRUE;

```

```

        // alarm(0);
    }
    break;
case BCC_OK:
    printf("BCC_OK\n");
    if (byte == FLAG)
    {
        printf("STOP\n");
        STOP = TRUE;
        state = START;
        if (c == C_UA)
            alarm(0);
    }
    else
        state = START;
    break;
}
// printf("%x\n", byte);
readbyte = byte;
return TRUE;
}

return FALSE;
}

int sendBuffer(unsigned char a, unsigned char c)
{
    // Create string to send
    unsigned char buf[5];

    buf[0] = FLAG;
    buf[1] = a;
    buf[2] = c;
    buf[3] = a ^ c;
    buf[4] = FLAG;

    return write(fd, buf, sizeof(buf));
}

// Alarm function handler
void alarmHandler(int signal)
{

```

```

    alarmEnabled = FALSE;
    alarmCount++;

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

////////////////////////////////////////
// LLOPEN
////////////////////////////////////////
int llopen(LinkLayer connectionParameters)
{
    printf("HEY\n");
    layer = connectionParameters;

    (void)signal(SIGALRM, alarmHandler);

    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;  // Blocking read until 5 chars received

```

```

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

// Now clean the line and activate the settings for the port
// tcflush() discards data written to the object referred to
// by fd but not transmitted, or data received but not read,
// depending on the value of queue_selector:
// TCIFLUSH - flushes data received but not read.
tcflush(fd, TCIOFLUSH);

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

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

// In non-canonical mode, '\n' does not end the writing.
// Test this condition by placing a '\n' in the middle of the
buffer.
// The whole buffer must be sent even with the '\n'.

// Set alarm function handler

int bytes = 0;

// Wait until all bytes have been written to the serial port
sleep(1);

STOP = FALSE;

switch (connectionParameters.role)
{
case LlTx:

    while (STOP == FALSE && alarmCount <
connectionParameters.nRetransmissions)
    {
        if (alarmEnabled == FALSE)
        {

```

```

        bytes = sendBuffer(A_T, C_SET);
        printf("%d bytes written\n", bytes);
        alarm(connectionParameters.timeout); // Set alarm to be
triggered

        alarmEnabled = TRUE;
        state = START;
    }

    stateMachine(A_T, C_UA, 0, 0);
}
alarm(0);

if (alarmCount >= layer.nRetransmissions)
{
    printf("ERRO TIME OUT\n");
    return -1;
}
printf("LLOPEN OK\n");
break;
case LlRx:
    while (STOP == FALSE)
    {
        stateMachine(A_T, C_SET, 0, 0);
    }
    bytes = sendBuffer(A_T, C_UA);
    printf("RESPONSE TO LLOPEN TRANSMITER. %d bytes written\n",
bytes);
    break;
default:
    break;
}

return 0;
}

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

int stuffing(const unsigned char *msg, int newSize, unsigned char
*stuffedMsg)
{

```

```

int size = 0;

stuffedMsg[size++] = msg[0];

printf("\nSTUFFING\n");

printf("%x\n", stuffedMsg[size - 1]);

for (int i = 1; i < newSize; i++)
{
    if (msg[i] == FLAG || msg[i] == ESCAPE)
    {
        stuffedMsg[size++] = ESCAPE;
        printf("%x\n", stuffedMsg[size - 1]);
        stuffedMsg[size++] = msg[i] ^ 0x20;
        printf("%x\n", stuffedMsg[size - 1]);
    }
    else
    {
        stuffedMsg[size++] = msg[i];
        printf("%x\n", stuffedMsg[size - 1]);
    }
}

printf("\nEND STUFFING\n");

return size;
}

int destuffing(const unsigned char *msg, int newSize, unsigned char
*destuffedMsg)
{
    int size = 0;

    printf("\nDESTUFFING\n");

    destuffedMsg[size++] = msg[0];
    printf("%x\n", destuffedMsg[size - 1]);

    for (int i = 1; i < newSize; i++)
    {
        if (msg[i] == ESCAPE)
        {

```

```

        destuffedMsg[size++] = msg[i + 1] ^ 0x20;
        printf("%x\n", destuffedMsg[size - 1]);
        i++;
    }
    else
    {
        destuffedMsg[size++] = msg[i];
        printf("%x\n", destuffedMsg[size - 1]);
    }
}

printf("\nEND DESTUFFING\n");
printf("size: %d\n", size);

return size;
}

unsigned char calculateBCC2(const unsigned char *buf, int dataSize, int
startingByte)
{
    if (dataSize < 0)
    {
        printf("Error buf Size: %d\n", dataSize);
    }
    unsigned char BCC2 = 0x00;
    for (unsigned int i = startingByte; i < dataSize; i++)
    {
        BCC2 ^= buf[i];
    }
    printf("Calculate BCC2: %x\n", BCC2);
    return BCC2;
}

int llwrite(const unsigned char *buf, int bufSize)
{
    int newSize = bufSize + 5; // FLAG + A + C + BCC1 + .... + BCC2 +
FLAG

    unsigned char msg[newSize];

    static int packet = 0;

    msg[0] = FLAG;

```



```

msg[1] = A_T;
msg[2] = C_INF(packet);
msg[3] = BCC(A_T, C_INF(packet));

unsigned char BCC2 = buf[0];
for (int i = 0; i < bufSize; i++)
{
    msg[i + 4] = buf[i];
    if (i > 0)
        BCC2 ^= buf[i];
}

msg[bufSize + 4] = BCC2;

unsigned char stuffed[newSize * 2];
newSize = stuffing(msg, newSize, &stuffed);
stuffed[newSize] = FLAG;
newSize++;

STOP = FALSE;
alarmEnabled = FALSE;
alarmCount = 0;
state = START;

int numtries = 0;

int reject = FALSE;

while (STOP == FALSE && alarmCount < layer.nRetransmissions)
{
    if (alarmEnabled == FALSE)
    {
        numtries++;
        bytes = write(fd, stuffed, newSize);
        printf("Data Enviada. %d bytes written, %d° try...\n",
bytes, numtries);
        alarm(layer.timeout); // Set alarm to be triggered
        alarmEnabled = TRUE;
        state = START;
    }

    stateMachine(A_T, NULL, 0, 1);
}

```

```

        if ((packet == 0 && response == REJ1) || (packet == 1 &&
response == REJ0))
        {
            reject = TRUE;
        }

        if (reject == TRUE)
        {
            alarm(0);
            alarmEnabled = FALSE;
        }
    }
    packet = (packet + 1) % 2;
    alarm(0);
    printf("Data Accepted!\n");

    return 0;
}

////////////////////////////////////
// LLREAD
////////////////////////////////////
int llread(unsigned char *buffer)
{
    int bytesread = 0;

    static int packet = 0;

    unsigned char stuffedMsg[MAX_BUFFER_SIZE];
    unsigned char unstuffedMsg[MAX_PACKET_SIZE + 7];

    STOP = FALSE;
    state = START;

    int bytes = 0;

    while (STOP == FALSE)
    {
        if (stateMachine(A_T, C_INF(packet), 1, 0))
        {
            stuffedMsg[bytesread] = readbyte;
            bytesread++;
        }
    }

```

```

}

printf("DATA RECEIVED\n");

int s = destuffing(stuffedMsg, bytesread, unstuffedMsg);

unsigned char receivedBCC2 = unstuffedMsg[s - 2];
printf("RECEIVED BCC2: %x\n", receivedBCC2);
unsigned char expectedBCC2 = calculateBCC2(unstuffedMsg, s - 2, 4);
printf("EXPECTED BCC2: %x\n", expectedBCC2);

if (receivedBCC2 == expectedBCC2 && unstuffedMsg[2] ==
C_INF(packet))
{
    packet = (packet + 1) % 2;
    sendBuffer(A_T, RR(packet));
    memcpy(buffer, &unstuffedMsg[4], s - 5);
    printf("STATUS 1 ALL OK: %x , %x\nSENDING RESPONSE\n",
receivedBCC2, unstuffedMsg[2]);
    return s - 5;
}
else if (receivedBCC2 == expectedBCC2)
{
    sendBuffer(A_T, RR(packet));
    tcflush(fd, TCIFLUSH);
    printf("Duplicate packet!\n");
}
else
{
    sendBuffer(A_T, REJ(packet));
    tcflush(fd, TCIFLUSH);
    printf("Error in BCC2, sent REJ\n");
}
return -1;
}

////////////////////////////////////////
// LLCLOSE
////////////////////////////////////////
int llclose(int showStatistics)
{
    STOP = FALSE;

```

```

alarmEnabled = FALSE;
alarmCount = 0;

state = START;
response = OTHER;

switch (getRole())
{
case LlTx:

    while (STOP == FALSE && alarmCount < getnTransmissions())
    {
        if (alarmEnabled == FALSE)
        {
            bytes = sendBuffer(A_T, DISC);
            printf("Sent DISC\n");
            printf("%d bytes written\n", bytes);
            alarm(getTimeOut()); // Set alarm to be triggered
            alarmEnabled = TRUE;
        }

        stateMachine(A_R, DISC, 0, 0);
    }
    if (alarmCount == getnTransmissions())
        return -1;
    printf("Received DISC\n");
    bytes = sendBuffer(A_R, C_UA);
    printf("Sent UA\n");
    printf("%d bytes written\n", bytes);
    break;
case LlRx:
    while (STOP == FALSE)
    {
        stateMachine(A_T, DISC, 0, 0);
    }
    printf("Received DISC\n");
    bytes = sendBuffer(A_R, DISC);
    printf("Sent DISC\n");
    printf("%d bytes written\n", bytes);
    STOP = FALSE;
    state = START;
    while (STOP == FALSE)
    {

```

```

        stateMachine(A_R, C_UA, 0, 0);
    }
    printf("Received UA\n");
    break;
default:
    break;
}

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

close(fd);

return 0;
}

```