

DECEMBER 6, 2021



# TRABALHO PRÁTICO Nº1

## REDES DE COMPUTADORES

ANDRÉ DE JESUS FERNANDES FLORES & TIAGO ANDRÉ BATISTA RODRIGUES  
FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

## Sumário

O trabalho foi desenvolvido no âmbito da Unidade Curricular de Redes de Computadores. O trabalho consiste em desenvolver um protocolo de ligação de dados fiável para a transmissão de ficheiros entre dois computadores.

O trabalho foi concluído com sucesso, tendo-se alcançado todos os objetivos explicitados.

## Introdução

O objetivo do projeto era desenvolver um protocolo de ligação de dados capaz de fornecer um serviço de comunicação fiável entre dois sistemas ligados por cabo série, de acordo com a especificação fornecida no guião. Este relatório visa documentar o funcionamento do protocolo implementado.

O relatório tem as seguintes unidades lógicas:

- Arquitetura
- Estrutura de Código
- Casos de Uso Principais
- Protocolo de Ligação Lógica
- Protocolo de Aplicação
- Validação
- Conclusão

## Arquitetura

O protocolo está dividido em dois blocos funcionais: o transmissor (*transmitter*) e o recetor (*receiver*). Ambos os blocos utilizam funções definidas na camada de aplicação e na camada de ligação de dados e estão definidos no mesmo ficheiro, havendo no entanto independência entre o funcionamento dos dois.

## Estrutura de Código

O código encontra-se dividido em três ficheiros: o *alarm.c*, que inclui funções responsáveis pelo tratamento de sinais **SIGALRM** usados para implementar o funcionamento de *timeouts*, o *linklayer.c*, que inclui funções responsáveis pela implementação do protocolo de ligação de dados e o *main.c*, responsável pelo protocolo de aplicação.

### *alarm.c*

#### Funções de tratamento de sinais

- **atend()** – imprime na consola que fez handle de um sinal, ativa uma *flag* que indica que ocorreu um alarme

### *linklayer.c*

#### Funções do protocolo de ligação de dados

- **llopen()** – estabelece a ligação entre o *transmitter* e o *receiver*, se for chamado no *transmitter* envia uma trama com *SET* e recebe uma trama com *UA*, se for chamado no *receiver* recebe uma trama *SET* e envia uma trama *UA*

- ***llclose()*** – termina a ligação, se for chamado no *transmitter*, envia a trama *DISC*, recebe a trama *DISC*, e envia a trama *UA*, se for chamado no *receiver*, recebe a trama *DISC*, envia a trama *DISC*, e recebe a trama *UA*
- ***llwrite()*** – efetua o *byte stuffing* nos pacotes de dados que recebe e envia-os numa trama de informação para o *receiver*
- ***llread()*** – recebe pacotes de dados e efetua *byte destuffing*
- ***read\_message()*** – lê uma trama de supervisão

*main.c*

#### Funções da camada de aplicação

- ***extract\_filename()*** – extrai um nome de ficheiro de um pacote de controlo
- ***make\_start\_packet()*** – cria um pacote de controlo com um nome fornecido
- ***transmit()*** – transmite o ficheiro indicado pelo argumento *path*
- ***receive()*** – recebe um ficheiro

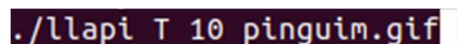
#### Variáveis globais

- ***int alarm\_flag***
- ***int alarm\_count***
- ***int state***
- ***struct termios oldtio***
- ***struct termios newtio***
- ***char llwrite\_start***
- ***char llread\_start***

### Casos de Uso Principais

#### *Interface*

A interface permite ao utilizador escolher em que modo está a utilizar o programa, *transmitter* (T) ou *receiver* (R), a porta série que está a utilizar e, no caso de modo *transmitter*, o ficheiro a enviar.



```
./llapi T 10 pinguim.gif
```

Figura 1 exemplo de comando para inicializar transmissão

#### *Sequência de eventos*

- O transmissor escolhe o ficheiro a ser enviado
- É estabelecida a ligação (ambos o transmissor e o recetor chamam *llopen()*)
- O transmissor envia os dados divididos em pacotes e o recetor escreve-os num ficheiro (o transmissor chama *llwrite()* e o recetor *llread()*)
- A ligação é terminada (ambos chamam *llclose()*)

### Protocolo de Ligação Lógica

No protocolo de ligação lógica foram implementadas as funções requeridas na especificação do projeto.

### *llopen()*

Função responsável por estabelecer a ligação entre os dois computadores. Abre e prepara a porta série, especifica a função de tratamento de sinais **SIGALRM**. Se for chamada no transmissor, envia uma trama de supervisão *SET* e aguarda o *UA* enviado pelo recetor tendo um *timeout* de três segundos até tentar outra vez e um número máximo de três tentativas. A função constrói a trama de supervisão que envia e lê respostas através da função *read\_message()*, que lê tramas *byte a byte*. Se for chamada no recetor, tenta ler uma trama de supervisão *SET* através da função *read\_message()* e, se tal acontecer, envia uma trama de supervisão *UA*.

### *llwrite()*

Função responsável por efetuar *byte stuffing* a pacotes de dados e de os enviar em tramas de informação, apenas chamada no transmissor. Calcula o *bit* de paridade do pacote de dados e guarda o numa *array* com o pacote de dados, faz *byte stuffing* sobre essa *array* e coloca o resultado no campo de dados de uma trama de informação. Envia a trama e aguarda resposta (*timeout* de três segundos), se receber uma trama de supervisão *RR* retorna o número de *bytes* enviados, se receber uma trama de supervisão *REJ* ou ocorrer *timeout* envia a trama de informação outra vez, com um máximo de três tentativas. Caso não consiga enviar a trama de informação, retorna -1.

### *llread()*

Função responsável por ler tramas de informação *byte a byte* e executar *destuffing*. Lê a trama de informação *byte a byte* e verifica o valor *BCC1* (se este valor estiver errado descarta a trama). Depois, faz *destuffing* do campo de dados e verifica o valor *BCC2*, se este valor estiver certo envia uma trama de supervisão *RR* e guarda os dados, caso contrário, envia uma trama de supervisão *REJ* e tenta ler a trama outra vez.

### *llclose()*

Função responsável por terminar a ligação de dados. Se for chamada no transmissor envia uma trama de supervisão *DISC*, espera por uma resposta na forma de uma trama de supervisão *DISC* e envia uma trama de supervisão *UA*. Se for chamada no recetor espera pela receção de uma trama de supervisão *DISC*, envia uma trama de supervisão *DISC* e recebe uma trama de supervisão *UA*. No final fecha a porta série usando o seu *file descriptor*.

### *Funções Auxiliares*

A única função auxiliar aqui implementámos foi a *read\_message()* esta função lê uma trama supervisão *byte a byte*.

## Protocolo de Aplicação

O protocolo de aplicação é implementado, principalmente, nas funções *transmit()* e *receive()*, chamadas pela função *main()* do emissor e do recetor, respetivamente.

### *transmit()*

Esta função é responsável por enviar o ficheiro especificado no seu argumento. Começa por criar um pacote de início usando o nome do ficheiro e de o enviar através da função *llwrite()*, depois divide o ficheiro em pacotes de dados identificados e numerados e envia-los um a um também pela função *llwrite()*. No final envia um pacote de fim pela função *llwrite()*.

### *receive()*

Esta função é responsável por receber um ficheiro. Funciona com uma máquina de estados. No início (estado *WAITING*), lê pacotes até encontrar um pacote de início. Quando tal acontecer, extrai o nome do ficheiro a receber do pacote de início e cria um novo ficheiro com esse nome para onde vão ser escritos os dados, altera de estado para *WRITING*. De seguida lê e guarda os dados recebidos por pacotes de dados até receber um pacote de fim, onde termina de escrever e retorna.

### *Funções Auxiliares*

Apenas existem duas funções auxiliares. A *extract\_filename()* que extrai o nome de ficheiro de um pacote de início, e a *make\_start\_packet()* que cria um pacote de início com o nome de um ficheiro.

## Validação

### *Testes efetuados*

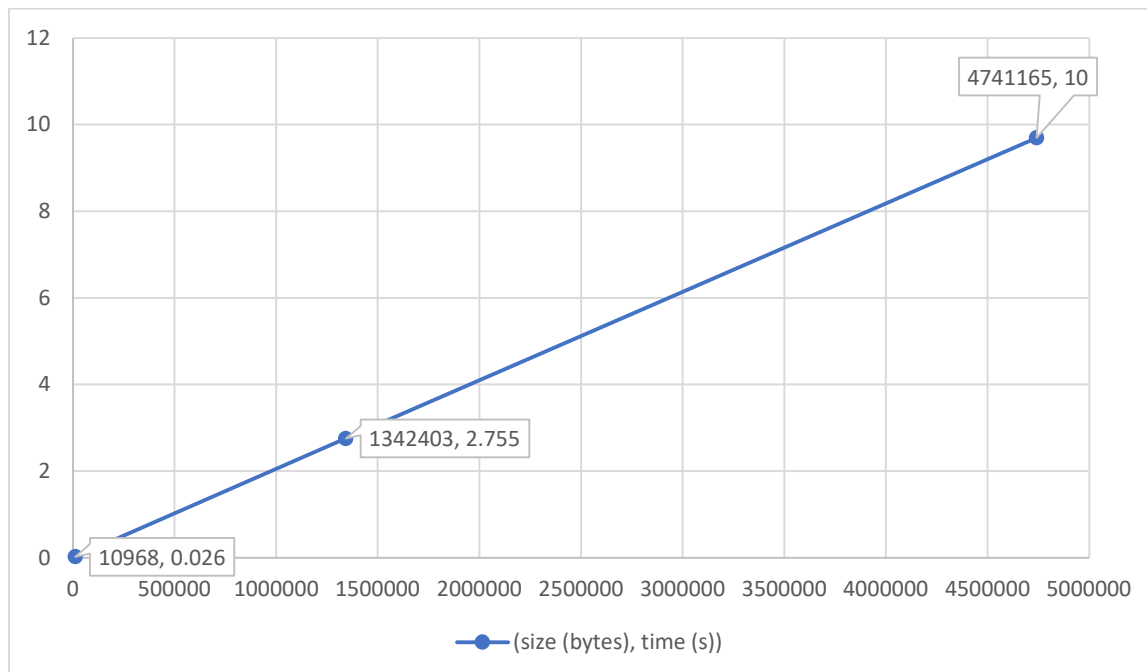
- Envio de vários ficheiros, com diferentes tamanhos.
- Interrupção e reconexão da ligação do cabo de série.
- Interrupção sem reconexão da ligação do cabo de série.

### *Resultados*

Todos os testes foram um sucesso, ocorrendo o comportamento esperado.

## Eficiência do Protocolo de Ligação de Dados

Para testar a eficiência do protocolo foram realizados testes de transmissão de ficheiros de vários tamanhos (10968 bytes, 1342403 bytes, 4741165 bytes) três vezes e registada a média do tempo de transmissão de cada um.



## Conclusão

### *Síntese*

Neste projeto foi implementado um protocolo de ligação de dados que fornece um serviço fiável na ligação de dois computadores.

### *Reflexão*

A concretização deste trabalho salientou a necessidade da independência de camadas, nenhuma das camadas precisa de ter conhecimento sobre funcionamento da outra.

## Anexo – Código Fonte

### main.c

```
/*Non-Canonical Input Processing*/

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

#include "linklayer.h"

#define _POSIX_SOURCE 1 /* POSIX compliant source */
#define FALSE 0
#define TRUE 1
#define PACKET_SIZE 256
#define WAITING 0 // Waiting for a file to be transmitted
#define WRITING 1 // Writing a file
#define DATA 0x01
#define START 0x02
#define END 0x03
#define C 0
#define N 1
#define L2 2
#define L1 3
#define D 4

volatile int STOP=FALSE;

int state;
int alarm_flag = 1;
```

```

int alarm_count = 0;

/**
 * Extract filename from a control packet
 *
 * @param packet Packet to read filename
 * @param filename Name of file
 */
void extract_filename(char * packet, char * filename) {
    int size = packet[1];

    for (int i = 0; i < size - 1; i++) {
        filename[i] = packet[2 + i];
    }
}

/**
 * Create a start control packet
 *
 * @param filename Name of file
 * @param packet Start packet
 * @return int Size of Packet
 */
int make_start_packet(char * filename, char * packet) {
    char size = (char) (strlen(filename) + 1);
    packet[0] = START;
    packet[1] = size;
    memcpy(packet + 2, filename, size);

    return size + 2;
}

/**
 * Transmit file at filename path through serial port indicated by fd
 *
 * @param fd File Descriptor
 * @param filename Name of file
 * @return int -1 in case of error, 0 otherwise
 */
int transmit(int fd, char * filename) {
    int fd_file;
    struct stat file_stat;
    char packet[1024];

```

```

int size = make_start_packet(filename, packet), n = 0;
char msg[256];

if((fd_file = open(filename, O_RDWR)) < 0) perror("Error opening file:
");

printf("Writing file: %s. Packet size: %d\n", filename, PACKET_SIZE);

// START PACKET
llwrite(fd, packet, size);

// DATA PACKETS
char read_data[PACKET_SIZE];
int read_size;

int c = 0;

while (true)
{
    if (true) {
        read_size = 0;

        for (; read_size < PACKET_SIZE; read_size++) {
            if (read(fd_file, read_data + read_size, 1) == 0) break;
        }

        if (read_size == 0) {
            printf("File over.\n");
            break;
        }

        packet[C] = DATA;
        packet[N] = c % 256;
        packet[L2] = (unsigned char) (read_size / 256);
        packet[L1] = (unsigned char) (read_size % 256);

        memcpy(packet + D, read_data, read_size);
    }

    if (llwrite(fd, packet, read_size + 5) < 0) return -1;
    c++;
}

```



```

        // END PACKET
        size = make_start_packet(filename, packet);
        packet[C] = END;

        llwrite(fd, packet, size);

        printf("Finished writing file\n");

        close(fd_file);

        return 0;
    }

/**
 * Receive a file through serial port indicate by file descriptor fd
 *
 * @param fd File Descriptor
 * @return int -1 in case of error , 0 otherwise
 */
int receive(int fd) {
    int fd_file, status = WAITING;

    while (true) {
        char packet[1024];
        int bytes_read = llread(fd, packet);

        // If state machine is waiting for file and receives start packet
        if (status == WAITING && packet[C] == START) {
            char filename[256];

            memset(filename, 0, 256);

            extract_filename(packet, filename);

            unlink(filename);

            if ((fd_file = open(filename, O_RDWR | O_CREAT, 0777)) < 0)
                perror("Error creating new file: ");

            status = WRITING;
        }

        // If state machine is writing and receives an end packet
        else if (status == WRITING && packet[C] == END) {
            printf("Finished receiving\n");

```

```

        close(fd_file);
        return 0;
    }

    // If state machine is writing and receives a data packet
    else if (status == WRITING && packet[C] == DATA) {
        unsigned char l2 = packet[L2], l1 = packet[L1];
        int res = l2 * 256 + l1;
        write(fd_file, packet + 4, res);
    }

    // Otherwise
    else {
        printf("Catastrophe!\n");
        close(fd_file);
        return -1;
    }
}

}

int main(int argc, char** argv)
{
    int fd, c, res, port;
    char buf[255];
    // Case: lesser arguments than should have
    if(argc < 3) {
        printf("./llapi T/R port_number [file to transfer]\n");
        exit(1);
    }

    // Get the transmitter/receiver state
    if (strcmp(argv[1], "T") == 0) state = TRANSMITER;
    else if (strcmp(argv[1], "R") == 0) state = RECEIVER;
    else {
        printf("Bad arguments!\n");
        printf("./llapi T/R port_number [file to transfer]\n");
        exit(-1);
    }

    if ((argc < 3 && state == RECEIVER) || (argc < 4 && state ==
TRANSMITER)) {
        printf("Bad arguments!\n");
        printf("./llapi T/R port_number [file to transfer]\n");
        exit(1);
    }
}

```

```

    sscanf(argv[2], "%d", &port);

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

    // Establish connection

    if ((fd = llopen(port, state)) < 0) exit(-1);
    printf("Establish connection\n");

    // Starting writing/reading packet from file
    if(state == TRANSMITER) {
        transmit(fd, argv[3]);
    } else {
        receive(fd);
    }

    // Close Connection
    llclose(fd);

    return 0;
}

```

## linklayer.h

```

#ifndef LLAPI_H
#define LLAPI_H

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

#include "alarm.h"

#define BAUDRATE B38400
#define RECEIVER 0x00

```

```

#define TRANSMITER 0x01
#define FLAG 0x7E
#define DISC 0x0B
#define SET 0x03
#define UA 0x07
#define ADDR 0x03
#define IADDR 1
#define ICTRL 2
#define IBCC1 3
#define RR(n) ((n << 7) | 0x05)
#define REJ(n) ((n << 7) | 0x01)

extern int alarm_flag;
extern int alarm_count;
extern int state;

/**
 * Establish the connection between 2 systems
 *
 * @param port Number of port
 * @param state State: Transmitter/Receiver
 * @return int -1 in case of error, 0 otherwise
 */
int llopen(int port, int state);

/**
 * Close connection
 *
 * @param fd File Descriptor
 * @return int -1 in case of error , 0 otherwise
 */
int llclose(int fd);

/**
 * Read from serial port to buffer according to protocol
 *
 * @param fd File Descriptor
 * @param buffer Buffer to write to
 * @return int
 */
int llread(int fd, char* buffer);

/**

```

```

* Write length number of bytes from buffer to serial port indicated by fd
*
* @param fd File Descriptor
* @param buffer Buffer to read from
* @param length NUmber of bytes from buffer to read
* @return int -1 in case of error, 0 otherwise
*/
int llwrite(int fd, char* buffer, int length);

/**
* Read supervision trama
*
* @param fd File Descriptor
* @param message message to write to
* @return int -1 in case of error, 0 otherwise
*/
int read_message(int fd, unsigned char * message);

#endif

```

## linklayer.c

```

#include "linklayer.h"

struct termios oldtio,newtio;

char llwrite_start = 0;
char llread_start = 1;

int llopen(int port, int state) {
    if (state != RECEIVER && state != TRANSMITER) return 1;

    char serial[256];
    snprintf(serial, 256, "/dev/ttyS%d", port);

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

    /*
    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.
    */

```

```

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

bzero(&newtio, sizeof(newtio));
newtio.c_cflag = 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]      = 100; /* inter-character timer unused */
newtio.c_cc[VMIN]       = 0; /* blocking read until 5 chars received */

/*
a  VTIME e VMIN devem ser alterados de forma a proteger com um temporizador
   leitura do(s) próximo(s) caracter(es)
*/

tcflush(fd, TCIOFLUSH);

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

unsigned char message[5] = {FLAG, ADDR, SET, SET ^ ADDR, FLAG};

alarm(0);

signal(SIGALRM, atend);

if (state == TRANSMITTER) {
    while (alarm_count < 3) {
        if (alarm_flag == 1) {

            write(fd, message, 5);
            alarm(3);

            alarm_flag = 0;
        }
    }
}

```

```

        if (!read_message(fd, message)) break;
        tcflush(fd, TCIOFLUSH);
    }

    if (alarm_count >= 3) {
        printf("llopen - timeout\n");
        return -1;
    }

    if ((message[ICTRL] ^ message[IADDR]) != message[IBCC1]) {
        printf("Parity error\n");
        return -1;
    }
}

else if (state == RECEIVER) {
    alarm_flag = 0;
    read_message(fd, message);
    if ((message[ICTRL] ^ message[IADDR]) != message[IBCC1]) {
        printf("Parity error\n");
        return -1;
    }

    message[ICTRL] = UA;
    message[IBCC1] = message[ICTRL] ^ message[IADDR];

    write(fd, message, 5);
} else {
    printf("Bad state!\n");
    return -1;
}

alarm(0);

alarm_count = 0;
alarm_flag = 0;

return fd; struct termios oldtio, newtio;

char llwrite_start = 0;
char llread_start = 1;
}

int llclose(int fd) {

```

```

if(state == TRANSMITER) {
    /*
        Write a DISC, read a DISC and write a UA to check if it is OK
    */
    char msg_tr[5] = {FLAG, ADDR, DISC, ADDR ^ DISC ,FLAG};
    char msg_re[5];
    write(fd, msg_tr, 5);
    read_message(fd, msg_re);

    if((msg_re[IADDR] ^ msg_re[ICTRL]) != msg_re[IBCC1]) {
        printf("BCC1 bad!\n");
        return -1;
    }
    char msg_ua[5] = {FLAG, ADDR, UA, ADDR ^ UA ,FLAG};

    write(fd, msg_ua, 5);

} else if (state == RECEIVER) {
    /*
        Read a DISC, write a DISC and receive a UA to check if it is OK
    */
    char msg_tr[5] = {FLAG, ADDR, DISC, ADDR ^ DISC ,FLAG};
    char msg_re[5];
    read_message(fd, msg_re);

    if((msg_re[IADDR] ^ msg_re[ICTRL]) != msg_re[IBCC1]) {
        printf("BCC1 bad!\n");
        return -1;
    }

    write(fd, msg_tr, 5);

    read_message(fd, msg_re);

    if((msg_re[IADDR] ^ msg_re[ICTRL]) != msg_re[IBCC1]) {
        printf("BCC1 bad!\n");
        return -1;
    }
} else {
    return -1;
}

printf("Closing FD!\n");
tcsetattr(fd,TCSANOW,&oldtio);

```



```

    close(fd);
    return 0;
}

int llread(int fd, char* buffer) {
    char *trama = (char *)malloc(sizeof(char)* 1024);
    char *stuffed = (char *)malloc(sizeof(char)* 1024);
    char *destuffed = (char *)malloc(sizeof(char)* 1024);
    int stuffed_size = 0, destuffed_size = 0, trama_index = 0;

    while (true)
    {
        trama_index = 0;
        stuffed_size = 0;
        destuffed_size = 0;
        while(true) {
            int res = read(fd, trama + trama_index, 1);

            if( trama[trama_index] != FLAG && trama_index == 0) continue;

            if( trama[trama_index] == FLAG && trama_index != 0) break;

            trama_index++;
        }

        // Check BCC1

        if ((trama[ICTRL] ^ trama[IADDR]) != trama[IBCC1]) {
            continue;
        }

        // Get stuffed data

        memcpy(stuffed, (trama + 4), trama_index - 4);

        stuffed_size = trama_index - 5;

        // Byte Destuffing

        for (int i = 0, j = 1; i < stuffed_size; i++, j++) {
            if (j == stuffed_size) destuffed[destuffed_size] = stuffed[i];
            else if (stuffed[i] == 0x7D && stuffed[j] == 0x5D) {
                destuffed[destuffed_size] = 0x7D;
                i++; j++;
            }
        }
    }
}

```

```

        else if (stuffed[i] == 0x7D && stuffed[j] == 0x5E) {
            destuffed[destuffed_size] = 0x7E;
            i++; j++;
        }
        else
            destuffed[destuffed_size] = stuffed[i];
        destuffed_size++;
    }

    // Check BCC2
    char xordata = destuffed[0];

    for(int i = 1; i < destuffed_size - 1; i++) {
        xordata = xordata ^ destuffed[i];
    }

    if (xordata != destuffed[destuffed_size - 1]) {
        unsigned char temp = REJ(llread_start);
        unsigned char msg[5] = {FLAG, ADDR, temp, ADDR ^ temp, FLAG};

        write(fd, msg, 5);
        continue;
    } else {
        unsigned char temp = RR(llread_start);
        unsigned char msg[5] = {FLAG, ADDR, temp, ADDR ^ temp, FLAG};

        write(fd, msg, 5);
        llread_start = llread_start ? 0 : 1;
        break;
    }
}

memcpy(buffer, destuffed, (destuffed_size-2) * sizeof(char));

free(destuffed);
free(stuffed);
free(trama);

return destuffed_size - 2;
}

int llwrite(int fd, char* buffer, int length) {
    char *unstuffed = (char *)malloc(sizeof(char) * 1024);
    char *stuffed = (char *)malloc(sizeof(char) * 1024);

```

```

char *trama = (char *)malloc(sizeof(char) * 1024);
int stuffed_index = 0;

// Fill Unstuffed

for (int i = 0; i < length; i++) unstuffed[i] = buffer[i];

// BCC2

char bcc2 = buffer[0];
for (int i = 1; i < length; i++) {
    bcc2 ^= buffer[i];
}

unstuffed[length] = bcc2;

// Stuff
for (int i = 0; i < length + 1; i++, stuffed_index++) {
    if (unstuffed[i] == 0x7E) {
        stuffed[stuffed_index++] = 0x7D;
        stuffed[stuffed_index] = 0x5E;
    }
    else if (unstuffed[i] == 0x7D) {
        stuffed[stuffed_index++] = 0x7D;
        stuffed[stuffed_index] = 0x5D;
    }
    else {
        stuffed[stuffed_index] = unstuffed[i];
    }
}

// Setup Trama
char temp_C = 0x00;
trama[0] = FLAG;
trama[1] = ADDR;
trama[2] = temp_C;
trama[3] = ADDR ^ temp_C;
memcpy(trama + 4 * sizeof(char), stuffed, (stuffed_index + 1) *
sizeof(char));
trama[5 + stuffed_index] = FLAG;

unsigned char ans[5];
int tries = 0;

// Write trama with ACK handling

```

```

while (tries < 3) {
    alarm(0);
    int temp = write(fd, trama, stuffed_index + 6);

    alarm(3);

    if (read_message(fd, ans) == 1) {
        printf("Read message failed\n");
        tries++;
        alarm_flag = 0;
        if (tries >= 3) exit(1);
        continue;
    }

    alarm(0);

    unsigned char t = (ans[IADDR] ^ ans[ICTRL]);

    if (t != ans[IBCC1]) {
        printf("BCC1 failed\n");
        tries++;
        continue;
    }

    t = REJ(llwrite_start ? 0 : 1);

    if (ans[ICTRL] == t) {
        printf("REJ\n");
        tcflush(fd, TCIOFLUSH);
        tries++;
        continue;
    }

    t = RR(llwrite_start ? 0 : 1);

    if (ans[ICTRL] == t) {
        llwrite_start = llwrite_start ? 0 : 1;
        break;
    }

    tries++;
}

free(trama);
free(unstuffed);

```

```

    free(stuffed);

    alarm(0);

    return tries >= 3 ? -1 : stuffed_index + 6;
}

int read_message(int fd, unsigned char * message) {
    int res, message_index = 0;
    while (alarm_flag == 0) {
        res = read(fd, message + message_index, 1);
        if(res < 0) continue;
        if (message[message_index] != FLAG && message_index == 0) continue;
        else if (message[message_index] == FLAG && message_index == 1) {
            message_index = 1;
            message[0] = FLAG;
            continue;
        }
        else if (message[message_index] == FLAG && message_index == 4) {
            message_index = 0;
            return 0;
        }
        else if (message_index >= 4) return -1;
        else {
            message_index++;
        }
    }
    return 1;
}

```

## alarm.h

```

#ifndef ALARM_H
#define ALARM_H

#include <stdio.h>
#include <unistd.h>

extern int alarm_flag;
extern int alarm_count;

```

```
/**
 * @brief Alarm handle
 *
 * /
void atend();

#endif
```

## alarm.c

```
#include "alarm.h"

void atend() {
    printf("Alarm #%d\n", ++alarm_count);
    alarm_flag = 1;
}
```