1º Trabalho Laboratorial:

Ligação de Dados

Redes de Computadores

Mestrado Integrado em Engenharia Informática e Computação

3º ano

Ricardo Fontão **up201806317**@fe.up.pt
Davide Castro **up201806512**@fe.up.pt

Turma 3 – Grupo 2

Table of Contents

1. Sumário....3
2. Introdução....3
3. Arquitetura....4
4. Estrutura do código....4
    Camada da aplicação....4
    Camada da ligação de dados....5
5. Casos de uso principais....6
6. Protocolo de ligação lógica....6
7. Protocolo de aplicação....7
8. Validação....8
9. Eficiência do protocolo de ligação de dados....9
    Variação do FER....9
    Variação do Baudrate....9
    Variação do tamanho dos pacotes de dados....10
    Variação de Tprop....10
10. Conclusão....10
Anexo I (código-fonte)....11
Anexo II (Dados de eficiência)....31

1. Sumário

Este trabalho foi realizado no âmbito da unidade curricular de Redes de Computadores e visa o desenvolvimento de uma aplicação simples e um protocolo de ligação de dados, para transferência de ficheiros através da Porta Série RS-232.

O projeto foi bem sucedido neste desenvolvimento, conseguindo uma transmissão de ficheiros segura e consistente mesmo perante situações como interrupção da ligação e ruído na transmissão.

2. Introdução

Com o trabalho laboratorial temos como objetivo desenvolver uma aplicação e implementar um protocolo de ligação de dados, para efetuar transferências de dados recorrendo à porta série com comunicação assíncrona.

Este relatório permite esclarecer a forma como foram implementadas as funcionalidades e o seu contexto teórico, assim como analisar os resultados obtidos. O relatório está assim organizado da seguinte forma:

- **Arquitetura**

  Principais blocos funcionais e interfaces usadas

- **Estrutura do código**

  Apresentação das estruturas de dados utilizadas, principais funções e APIs

- **Casos de uso principais**

  Identificação dos principais casos de uso e sequências de chamadas de funções

- **Protocolo de ligação lógica**

  Identificação das principais funcionalidades e formas de implementação aplicadas

- **Protocolo de aplicação**

  Identificação das principais funcionalidades e formas de implementação aplicadas

- **Validação**

  Descrição dos testes efetuados à aplicação e protocolo desenvolvidos, apresentando os resultados obtidos

- **Conclusão**

3. Arquitetura

O projeto está dividido em duas partes distintas: a camada da aplicação e a da ligação de dados. Com esta arquitetura é possível as duas camadas serem usadas para a transferência de dados, permitindo que cada uma das camadas não possua nenhum conhecimento de detalhes internos da outra, sendo assim independentes.

A camada de ligação de dados é responsável por tratar do estabelecimento da ligação entre o emissor e o recetor de dados, gerindo o fluxo da ligação através de mensagens de verificação e rejeição.

Já a camada de aplicação tem como função gerir o envio e receção dos ficheiros a transferir, utilizando chamadas à camada de ligação de dados. É nesta camada que o ficheiro é dividido em pacotes pelo emissor para serem enviados e recebidos pela aplicação do recetor.

4. Estrutura do código

Camada da aplicação

O código da aplicação está contido em dois ficheiros: application.c e application.h, que contêm todas as funções usadas pela a aplicação em modo emissor e recetor.

Estrutura de dados usada para armazenar os dados da aplicação:

```
typedef struct {
	int fileDescriptor;
	int status; /*SENDER | RECEIVER*/
	FILE* file;
} applicationLayer;
```

Nesta struct são guardados o file descriptor do ficheiro origem/destino dos dados, o modo a ser usado (recetor ou emissor), e o apontador para a estrutura tipo FILE com o ficheiro atual carregado.

Funções principais:

```
int set_port(char* port);
int close_port();
int llopen(char* port, int mode);
int llwrite(int fd, char* buffer, int length);
int llread(int fd, char* buffer);
int llclose(int fd);
int send_control(int type, char *filename, long fileSize);
int read_control(char* ctl, char* fileName, int* control_size);
int main(int argc, char **argv);
```

É na função **main** que o programa recebe como argumentos o path da porta série e também o modo atual, estando definidas as outras configurações como o *baudrate* e o tamanho máximo de um fragmento de dados nos ficheiros header como macros. A porta é depois definida na função **set_port** e fechada com **close_port** no final da execução do programa.

As funções **llopen**, **llwrite**, **llread** e **llclose** são aquelas que interagem diretamente com o protocolo de ligação de dados, sendo usadas para notificar o protocolo de quando abrir ou fechar a ligação, assim como enviar ou receber dados.

O código desta camada está contido nos ficheiros read.c, write.c e os seus respetivos headers, read.h e write.h. Cada um dos ficheiros corresponde aos diferentes modos, recetor e emissor, e contém as funções necessárias para cada um. Existem também os ficheiros state_machine.c, state_machine.h, common.c e common.h, que contêm funções usadas por ambos os modos.

Camada da ligação de dados

Funções principais:

- read:

```
int read_frame_reader(int fd, char* data, frame_type frame_type);
void read_set(int fd);
void read_disc(int fd);
int read_info(int fd, char* buffer);
int send_disc_receiver(int fd);
```

-write:

```
int send_set(int fd);
int send_disc_sender(int fd);
int send_info(int fd, char* data, int length);
void read_frame_writer(int fd, char* out, frame_type frame_type);
void send_info_frame(int fd, char* data, size_t size, int resend);
```

-common e state_machine:

```
void send_frame(int fd, frameType type);
STATE machine(unsigned char input, machine_type type, frame_type
frame_type);
```

Nesta camada são usadas várias funções para ler e enviar informação na porta, começadas por '**read**' ou '**send**', para cada tipo de informação, como *set's* ou pacotes('info'). A função **machine** é usada para ler quaisquer dados da porta série usando a uma máquina de estados, de forma a interpretá-los detetando erros de receção.

5. Casos de uso principais

A nossa aplicação tem dois casos de uso principais: **recetor**, cuja função é ler e enviar o ficheiro recebido através dos argumentos da linha de comando e **emissor**, que tem como função receber e guardar o ficheiro.

Para a especificação de cada modo de utilização fazemos use do argumentos da linha de comando:

./application -M read/write -P port -F file

Para cada flag:
- M – Usada para especificar o modo de uso (read/Read/R/r)(write/Write/w/W),
- P – Usada para especificar a porta-série a usar (ex: /dev/ttyS10)
- F – Flag que apenas deve ser definida caso estejamos em modo de escritor de forma a definir que ficheiro deve ser enviado.

Relativamente ao funcionamento de cada um dos modos:
No caso do emissor:
- Chamada de **llopen** em que é estabelecida a ligação com o **recetor**
- Abertura do ficheiro com recurso à função **openFile**
- Envio do pacote de controlo inicial com recurso à função **send_control**
- Leitura em bloco do ficheiro com a função **readFileBytes** e envio com recurso a **llwrite**
- Envio do pacote de controlo final, de novo com a função **send_control**
- Por fim, fecho da ligação com recurso a **llclose**

No caso do recetor:
- Chamada de **llopen** em que é estabelecida a ligação com o **emissor**
- Leitura de pacote de controlo inicial com recurso à função **read_control**
- Criação do ficheiro com recurso à função **openFile**
- Receção de pacotes com a função **llread** e escrita para ficheiro com **writeFileBytes**
- Receção do pacote de controlo final e verificação com a função **check_control**
- Por fim, fecho da ligação com recurso a **llclose**

6. Protocolo de ligação lógica

Na ligação de dados, uma das duas camadas que constituem o projeto, é onde são feitas as operações de mais baixo nível sobre os dados e a sua transferência pela porta série.

Nesta camada existem várias funcionalidades essenciais para a ligação:
- Estabelecer a ligação e terminá-la, com o envio e receção das tramas SET, UA e DISC;
- Encapsular os dados numa trama tipo I e enviá-los, ou desencapsular ao recebê-los;
- Validar tramas recebidas, enviando respostas de aceitação ou rejeiçao (RR e REJ) para o emissor;
- Assegurar que as tramas são sempre enviadas e recebidas corretamente, reenviando dados quando necessário.

Quanto à implementação do protocolo neste projeto este está dividido em duas partes: a do emissor e a do recetor (read e write).

Relativamente à secção do recetor, as funções principalmente usadas são **read_set**, **read_disc** e **read_info** que permitem a receção de tramas SET, DISC e I, respetivamente. Todas estas funções alocam espaço para a nova trama e chamam a função **read_frame_reader** com argumentos dependentes do tipo de trama de cada. Esta função tem a responsabilidade de ler, byte a byte, novos dados pela porta série, fazer *de-stuffing* destes e reconstruir a trama enviada pelo emissor, prevendo erros de receção usando uma máquina de estados, a função **machine**. No caso na receção de uma trama de informação, é efetuada uma verificação do BCC2 e é assim enviada a mensagem de resposta.

Para enviar as mensagens de resposta para o emissor ao receber dados recorremos às funções **send_rr** e **send_rej**, que enviam as tramas RR e REJ, dependendo da validade da trama recebida. Também é usada a função **send_disc_receiver** para o envio da trama DISC e receção da trama UA por parte do emissor no fim da ligação.

Por outro lado, na secção do emissor, usasmos as funções **send_se**t, **send_disc_sender** para o envio dos vários tipos de trama. Estas funções ativam um alarme e chamam a função **send_frame** junto com o seu tipo. Este processo é repetido periodicamente de acordo com o alarme e o número máximo de tentativas permitidas até ser recebida uma resposta apropriada por parte do recetor, com o uso da função **read_frame_writer**, que funciona de maneira semelhante da função read_frame_reader do recetor. Por sua parte, a função send_frame fica responsabilizada por construir a trama a enviar de acordo com o seu tipo, atribuindo os valores corretos de A (campo de endereço), C (campo de controlo) e BCC (campo de proteção).

Já no envio de tramas de informação, a função **send_info** é responsável por enviar dados ou reenviá-los quando é recebida uma mensagem de rejeição, fazendo uso de uma buffer auxiliar que guarda os dados enviados mais recentemente, uma flag de reenvio e também um alarme. Nesta função é chamada a função **send_info_frame**, que constrói a trama efetuando o *stuffing* dos dados, calcula os BCCs e envia pela porta.

7. Protocolo de aplicação

Esta outra camada de mais alto nível, a aplicação, é aquela que interage de forma direta com os ficheiros a transmitir, usando o protocolo de ligação de dados. Aqui existem as seguintes funcionalidades:

- Receber o input pelo utilizador, nomeadamente o modo de execução, o caminho para a porta série e, no caso do modo emissor, o nome do ficheiro;
- Abrir o ficheiro e ler os dados para enviar no modo emissor, ou escrever os dados recebidos no modo recetor;
- Efetuar a construção dos pacotes de controlo e de dados preparados para envio para o protocolo de ligação;
- Configurar inicialmente a porta série;
- Indicar quando abrir e fechar a ligação.

De forma a conseguirmos implementar estas funcionalidades criamos as funções **llopen**, **llread**, **llwrite** e **llclose**, que executam chamadas ao protocolo de aplicação e permitem, respetivamente, a abertura da ligação, leitura de dados, escrita de dados e fecho da aplicação. A aplicação trabalha principalmente com dois tipos de dados: pacotes de controlo e pacotes de dados.

Relativamente aos pacotes de controlo, estes são enviados pelo emissor, recorrendo à função **send_control**, que trata de construir o pacote usando os dados necessários (tamanho e nome do ficheiro) através da função **assemble_control_packet**. No lado do recetor é usada a função **read_control** para verificar estes pacotes, recebidos pela llread na função main. É também usada a função check_control para verificar a validade do pacote de controlo final.

Em relação aos pacotes de dados, no recetor são efetuadas leituras pela llread num ciclo na função **main** até ser encontrado o pacote de controlo final. Após cada leitura, se for verificado que o pacote recebido corresponde a um pacote de dados, é verificado se o número de sequência está correto e separa-se os dados do cabeçalho do pacote para os enviar para o ficheiro através da função **writeFileBytes**, pacote a pacote. No caso de um número de sequência inesperado, o programa termina com erro. O emissor, por outro lado, efetua a leitura de uma fracção do ficheiro (**readFileBytes**) e constrói um pacote, adicionando um cabeçalho aos dados com a função **assemble_data_packet**, usando de seguida a llwrite para enviar o pacote. Este procedimento é repetido até o ficheiro tiver sido enviado por completo.

No final, a aplicação fica responsável por notificar o protocolo de ligação de dados do fim da ligação, através da função llclose.

8. Validação

Para testar a aplicação e o protocolo de ligação desenvolvidos, foram-lhe aplicados vários testes, de forma a conseguir tirar conclusões sobre a eficácia do programa:

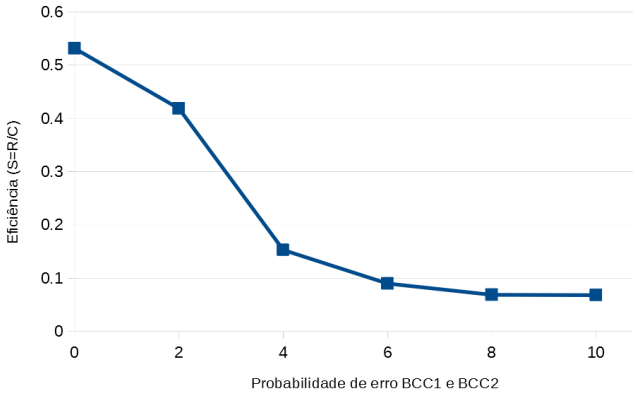
- Envio de ficheiros com tamanhos variados e de diferentes tipos, nomeadamente imagens, documentos e vídeos com tamanhos entre aproximadamente 10 kbytes e 200 Mbytes;
- Múltiplas interrupções na ligação da porta durante transmissões, desligando e ligando a porta série e, assim, testando a capacidade do protocolo de usar o alarme e reenviar os dados até ser obtida uma resposta aquando a retoma da ligação;
- Geração de ruído na porta série durante a transmissão, para testar se o protocolo é capaz de verificar a validade dos dados recebidos e enviar uma resposta REJ ao receber dados inválidos, ou, no emissor, reenviar os dados até receber uma mensagem RR;
- Envio de ficheiros usando diferentes tamanhos de pacotes entre 30 bytes e 65000 bytes, para verificar a modularidade da aplicação e do protocolo face a pacotes de tamanhos irregulares;
- Envio com variadas *baudrates*.

9. Eficiência do protocolo de ligação de dados

De forma a testar e avaliar a eficiência da nossa implementação do protocolo de ligação de dados realizámos três testes diferentes:

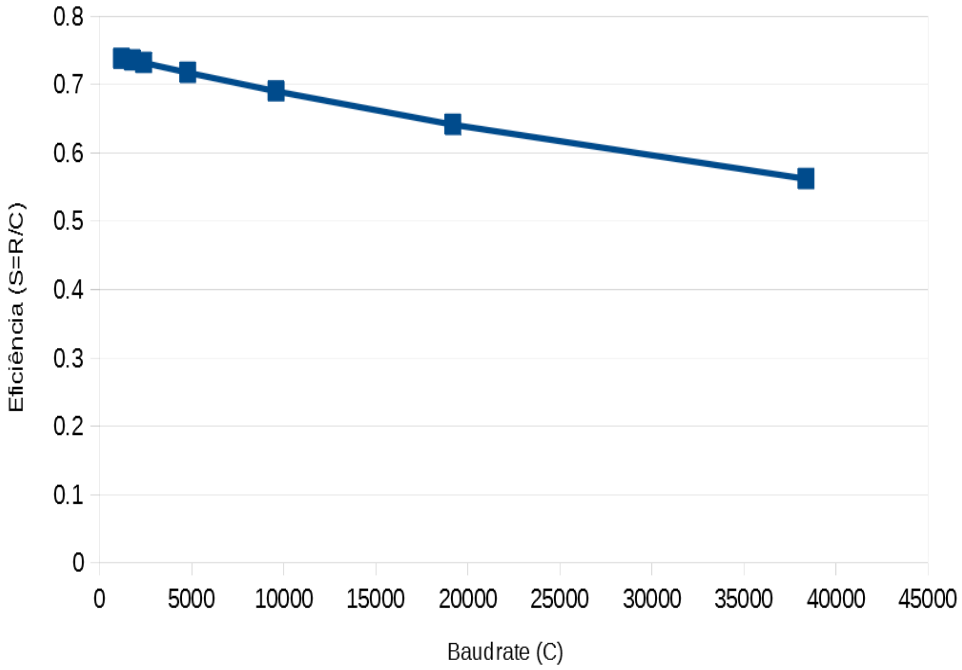
Variação do FER:

Neste teste variamos a probabilidade de erro na transmissão de pacotes. Vendo o gráfico podemos constatar que com o aumento desta probabilidade, a performance do protocolo diminiu substancialmente. Este teste foi realizado com um tamanho de pacote de dados de 100 bytes e um *baudrate* de 38400.



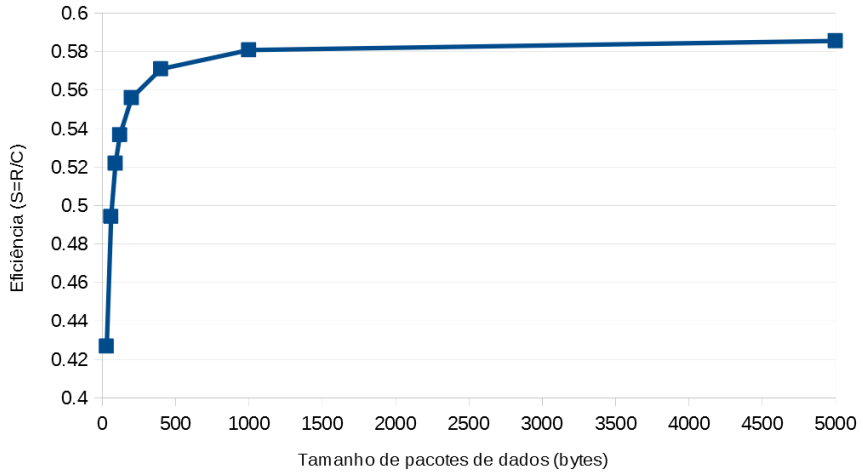
Variação do Baudrate:

A partir deste teste podemos concluir que com o aumento de *baudrate* (C) a eficiência do protocolo também aumenta. Para este teste foi usado um tamanho de pacote de 250 bytes.



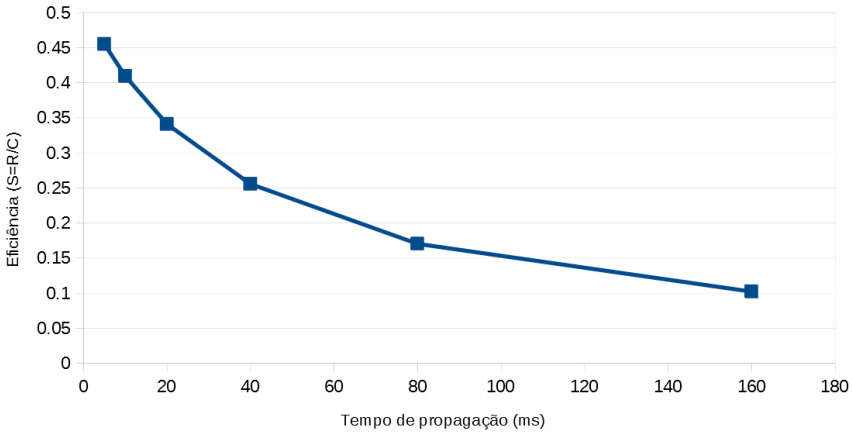
Variação do tamanho dos pacotes de dados:

Como seria de esperar a eficiência do protocolo aumento com o aumento do tamanho possível dos pacotes de dados. Neste teste foi usado um baudrate de 38400.



Variação de Tprop:

Neste teste variámos o teste de propagação e verificámos qual o seu impacto na eficiência do protocolo:



10. Conclusão

O projeto centra-se na implementação de um serviço de transmissão fiável de ficheiros usando a porta série, conseguindo a independência entre a camada que efetua a interpretação dos dados, a aplicação, e aquela responsável pela ligação entre o emissor e o recetor assegurando coesão na ligação, o protocolo de ligação de dados.

O grupo considera que todos os objetivos pretendidos foram alcançados possuindo agora uma melhor compreensão de alguns conceitos teóricos da ligação de dados.

Anexo I (código-fonte)
Application.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include "application.h"
#include "message_macros.h"
#include "read.h"
#include "write.h"
#include "common.h"

applicationLayer app;
struct termios oldtio, newtio;


int openFile(const char *name, int mode){
        if(mode == RECEIVER){
                app.file = fopen(name, "wb"); //HARDCODE
        } else if(mode == SENDER){
                app.file = fopen(name, "rb");
        } else {
                printf("Invalid mode\n");
                return -1;
        }
        if(app.file == NULL){
                printf("Failed to open file\n");
                return -1;
        }
        return 0;
}

long readFileBytes(char* result, long size_to_read){
        long len;
        if((len = fread(result, 1, size_to_read, app.file)) == 0){
                perror("Failed to read from file\n");
                return -1;
        }
    return len;
}

long readFileInfo(){
        struct stat buf;
        int fd = fileno(app.file);
        fstat(fd, &buf);
        return buf.st_size;
}

int writeFileBytes(char* data, long size){
        if(fwrite(data, 1, size, app.file) < 0){
                perror("Error writing to file\n");
                return -1;
        }
    return 0;
}

int send_control(int type, char *filename, long fileSize) {
        char packet[256];

        int size = assemble_control_packet(type, filename, fileSize, packet);

        if(size > MAX_PACKET_SIZE) {
                printf("Control size cannot be larger than maximum packet size\n");
                return -1;
        }

        if(llwrite(app.fileDescriptor, packet, size) == -1) {
                printf("Error sending control packet\n");
                return -1;
```

```
        }

        return 0;
}

int assemble_data_packet(char* data, int length, int sequenceN, char* packet) {

        packet[0] = C_DATA;
        packet[1] = sequenceN % 256;
        packet[2] = length / 256;
        packet[3] = length % 256;

        int i = 0;
        for(; i < length; ++i) {
                packet[i + 4] = data[i];
        }

        return i + 4;
}

int assemble_control_packet(int type, char* filename, long fileSize, char* packet) {

        if(type == START_C)
                packet[0] =     C_START;
        else
                packet[0] = C_END;

        packet[1] = FILE_SIZE;
        packet[2] = sizeof(fileSize);

        int k = 3;
        int j = sizeof(fileSize) - 1;
        for(; k < sizeof(fileSize) + 3; k++) {
                packet[k] = (fileSize >> (j*8)) & 0xff;
                j--;
        }

        packet[k++] = FILE_NAME;
        packet[k++] = strlen(filename);

        int i = 0;
        for(; i < strlen(filename); ++i) {
                packet[k + i] = filename[i];
        }


        //Other way around

        // packet[1] = FILE_NAME;
        // packet[2] = strlen(ff);
        // int i = 0;
        // for(; i < strlen(ff); ++i) {
        //      packet[3 + i] = ff[i];
        // }

        // packet[i + 3] = FILE_SIZE;
        // packet[i + 4] = sizeof(fileSize);
        // packet[i + 5] = (fileSize >> 24) & 0xff;
        // packet[i + 6] = (fileSize >> 16) & 0xff;
        // packet[i + 7] = (fileSize >> 8) & 0xff;
        // packet[i + 8] = fileSize & 0xff;

        return k + i;
}

int llopen(char* port, int mode) {
        printf("%s", port);
    int fd = set_port(port);

    if(mode == SENDER) {
        if(send_set(fd) < 0)
                        return -1;
    } else if(mode == RECEIVER) {
        read_set(fd);
    }

        printf("---Connection established---\n");
```

```
    return fd;
}

int llread(int fd, char* buffer){
        return read_info(fd, buffer);
}

int llwrite(int fd, char* buffer, int length){
        return send_info(fd, buffer, length);
}

int llclose(int fd){

        if(app.status == SENDER) {
        if(send_disc_sender(fd) < 0)
                        return -1;
    } else if(app.status == RECEIVER) {
        read_disc(fd);
                if(send_disc_receiver(fd) < 0)
                        return -1;
    }

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

        if(close_port() < 0){
                printf("Error closing port\n");
                exit(-1);
        }

    return 0;
}

int set_port(char* port){

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

        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] = 0; /* inter-character timer unused */
        newtio.c_cc[VMIN] = 1;  /* blocking read until 5 chars received */

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

        tcflush(fd, TCIOFLUSH);

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

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

        return fd;
}

int close_port(){
        sleep(1);
        if(tcsetattr(app.fileDescriptor, TCSANOW, &oldtio) < 0){
                return -1;
        }
```

```
        if(close(app.fileDescriptor) < 0){
                return -1;
        }

        return 0;
}

int read_control(char* ctl, char* fileName, int* control_size){
        if((*control_size = llread(app.fileDescriptor, ctl)) < 0){
                printf("--Error reading--\n");
                free(ctl);
                exit(-1);
        }

        char* control = ctl;

        if(control[0] != C_START && control[0] != C_END){
                printf("Invalid control\n");
                exit(-1);
        }

        long file_size = 0;
        char size[4];
        bzero(size, 4);
        if(control[1] == FILE_SIZE){
                int block_size = control[2];

                if(3 + block_size > MAX_PACKET_SIZE) {
                        printf("Control is larger than max size\n");
                        return -1;
                }

                int j = block_size - 1;
                for(int i = 0; i < block_size; i++){
                        int part = ((unsigned char)(control[i+3]) << (j * 8));
                        file_size |= part;
                        j--;
                }

                int name_size = control[block_size + 4];
                if(block_size + 4 + name_size > MAX_PACKET_SIZE) {
                        printf("Control is larger than max size\n");
                        return -1;
                }

                for(int i = 0; i < name_size; i++){
                        fileName[i] = control[i + block_size + 5];
                }
        } else if(control[1] == FILE_NAME){
                int name_size = control[2];

                if(3 + name_size > MAX_PACKET_SIZE) {
                        printf("Control is larger than max size\n");
                        return -1;
                }

                for(int i = 0; i < name_size; i++){
                        fileName[i] = control[i + 3];
                }

                int block_size = control[name_size + 4];

                if(name_size + 4 + block_size > MAX_PACKET_SIZE) {
                        printf("Control is larger than max size\n");
                        return -1;
                }

                int j = block_size - 1;
                for(int i = 0; i < block_size; i++){
                        file_size |= ((unsigned char)(control[name_size + i + 5]) << (j * 8));
                        j--;
                }
        }

        return file_size;
}
```

```
int check_control(char* control, char* buffer, int size){
        //print_frame(control, size);
        //print_frame(buffer, size);


        for(int i = 1; i < size; i++){
                if(control[i] != buffer[i]){
                        return -1;
                }
        }

        return 0;
}

int main(int argc, char **argv) {
        int c;
        int file_is_set = 0, mode_is_set = 0, port_is_set = 0;
        char file_name[1024] = "";
        char port_name[1024] = "";

        while ((c = getopt(argc, argv, "F:P:M:")) != -1) {
                switch (c) {
                case 'F':
                        strcpy(file_name, optarg);
                        file_is_set = 1;
                        break;
                case 'P':
                        strcpy(port_name, optarg);
                        port_is_set = 1;
                        break;
                case 'M':
                        if(strcmp(optarg, "read") == 0 || strcmp(optarg, "Read") == 0 || strcmp(optarg,
"R") == 0 || strcmp(optarg, "r") == 0){
                                app.status = RECEIVER;
                        } else if(strcmp(optarg, "write") == 0 || strcmp(optarg, "Write") == 0 ||
strcmp(optarg, "W") == 0 || strcmp(optarg, "w") == 0){
                                app.status = SENDER;
                        } else {
                                printf("Usage : %s -M read/write -P port -F file (only when write mode is
specified)\n", argv[0]);
                                exit(-1);
                        }
                        mode_is_set = 1;
                        break;
                case '?': // invalid option
                        printf("Usage : %s -M read/write -P port -F file (only when write mode is
specified)\n", argv[0]);
                        exit(-1);
                        break;
                default:
                        printf("Usage : %s -M read/write -P port -F file (only when write mode is
specified)\n", argv[0]);
                        exit(-1);
                        break;
                }
        }

        if(port_is_set == 0){
                printf("Usage : %s -M read/write -P port -F file (only when write mode is specified)\n",
argv[0]);
                exit(-1);
        }

        if(mode_is_set == 0){
                printf("Usage : %s -M read/write -P port -F file (only when write mode is specified)\n",
argv[0]);
                exit(-1);
        }

        if((file_is_set == 0) && app.status == SENDER){
                printf("Usage : %s -M read/write -P port -F file (only when write mode is specified)\n",
argv[0]);
                exit(-1);
        }

        if((file_is_set == 1) && app.status == RECEIVER){
                printf("Usage : %s -M read/write -P port -F file (only when write mode is specified)\n",
argv[0]);
```

```
                exit(-1);
        }

        if((app.fileDescriptor = llopen(port_name, app.status)) < 0){
                printf("Failed to open file\n");
                exit(-1);
        }

        if(app.status == RECEIVER){

                char control[MAX_PACKET_SIZE];
                char buffer[MAX_PACKET_SIZE];
                int read_size;
                int control_size;

                // int file_size = read_control(&control, file_name);
                read_control(control, file_name, &control_size);

                char file_buffer[MAX_CHUNK_SIZE];

                if(openFile(file_name, app.status) < 0){
                        perror("Error opening file\n");
                        exit(-1);
                }

                int control_found = 0;
                int sequenceN = 0;

                while(control_found == 0){
                        if((read_size = llread(app.fileDescriptor, buffer)) < 0){
                                printf("--Error reading--\n");
                                fclose(app.file);
                                exit(-1);
                        }

                        if(buffer[0] == C_END){
                                //End buffer is different from start buffer
                                if(check_control(control, buffer, control_size) < 0){
                                        printf("--End control packet is different from start control
packet--\n");

                                        fclose(app.file);
                                        exit(-1);
                                }
                                control_found = 0;
                                break;
                        }

                        //Check for sequence number
                        if((unsigned char)(buffer[1]) != sequenceN){
                                printf("--Received packet with wrong sequence number. Aborting--\n");
                                fclose(app.file);
                                exit(-1);
                        }

                        int packet_size = (unsigned char)(buffer[2]) * 256 + (unsigned char)(buffer[3]);
                        // printf("PACKET SIZE: %d\n", packet_size);

                        for(int i = 4; i < packet_size + 4; i++){
                                file_buffer[i - 4] = buffer[i];
                        }
                        //printf("FILE DATA: %s\n", file_buffer);
                        writeFileBytes(file_buffer, packet_size);

                        sequenceN = (sequenceN + 1) % 255;
                }

                fclose(app.file);

                if(llclose(app.fileDescriptor) < 0){
                        printf("Failed closing\n");
                        exit(-1);
                }

        } else if(app.status == SENDER) {

                if(openFile(file_name, app.status) < 0){
                        perror("Error opening file\n");
                        exit(-1);
```

```
                }

                long file_size = readFileInfo();
                char file[MAX_CHUNK_SIZE];

                if(send_control(START_C, file_name, file_size) < 0) {
                        printf("Error sending control packet\n");
                        fclose(app.file);
                        exit(-1);
                }

                int size_remaining = file_size;
                int size_to_send;
                int sequenceN = 0;

                while(size_remaining > 0){

                        if(size_remaining < MAX_CHUNK_SIZE){
                                if((size_to_send = readFileBytes(file, size_remaining)) < 0){
                                        perror("Error reading file\n");
                                        fclose(app.file);
                                        exit(-1);
                                }
                        } else {
                                if((size_to_send = readFileBytes(file, MAX_CHUNK_SIZE)) < 0){
                                        perror("Error reading file\n");
                                        fclose(app.file);
                                        exit(-1);
                                }
                        }

                        size_remaining -= size_to_send;

                        char* packet = (char*)malloc(size_to_send + 4);
                        int packet_size = assemble_data_packet(file, size_to_send, sequenceN, packet);
                        // printf("PACKET SIZE %d\n", packet_size);

                        if(llwrite(app.fileDescriptor, packet, packet_size) < 0){
                                printf("--Error writing--\n");
                                free(packet);
                                fclose(app.file);
                                exit(-1);
                        }

                        sequenceN = (sequenceN + 1) % 255;
                        free(packet);
                }

                //ler dados do ficheiro e chamar send_data()
                send_control(END_C, file_name, file_size);
                printf("Sent Control\n");

                fclose(app.file);

                if(llclose(app.fileDescriptor) < 0){
                        printf("Failed closing\n");
                        exit(-1);
                }
        }

        return 0;
}
```

Application.h

```
#ifndef _APPLICATION_H
#define _APPLICATION_H

#define SENDER      0
#define RECEIVER    1

#define BAUDRATE B38400
#define _POSIX_SOURCE 1 /* POSIX compliant source */

#define FALSE 0
#define TRUE 1

#define START_C 0
#define END_C 1

#define C_DATA  0x01
#define C_START 0x02
#define C_END   0x03

#define FILE_SIZE    0
#define FILE_NAME    1

typedef struct
{
    int fileDescriptor;
    int status; /*SENDER | RECEIVER*/
    FILE* file;
} applicationLayer;

int set_port(char* port);
int close_port();

int llopen(char* port, int mode);
int llwrite(int fd, char* buffer, int length);
int llread(int fd, char* buffer);
int llclose(int fd);

int assemble_control_packet(int type, char *filename, long fileSize, char* packet);
int assemble_data_packet(char* data, int length, int sequenceN, char* packet);

int send_control(int type, char *filename, long fileSize);
int read_control(char* ctl, char* fileName, int* control_size);
int check_control(char* control, char* buffer, int size);

int openFile(const char *name, int mode);
long readFileBytes(char* result, long size_to_read);
long readFileInfo();
int writeFileBytes(char* data, long size);
#endif
```

common.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include "common.h"
#include "message_macros.h"


void write_to_port(int fd, char* data, size_t s){
        int sent = write(fd, data, s);
        printf("%d bytes written\n", sent);
}

void print_frame(char* frame, size_t s){
    for(int i = 0; i < s; i++){
            printf(" %x ", (unsigned char)(frame[i]));
        }
    printf("\n");
}
```

```c
void send_frame(int fd, frameType type){

    if(type == SET){
        char set_frame[5];
            set_frame[0] = FLAG;
        set_frame[1] = A_SENDER;
        set_frame[2] = C_SET;
        set_frame[3] = A_SENDER ^ C_SET;
        set_frame[4] = FLAG;

        write_to_port(fd, set_frame, 5);
        printf("Sent SET frame : ");
        print_frame(set_frame, 5);
    }

    if(type == UA_RECEIVER){
        char ua_frame[5];
        ua_frame[0] = FLAG;
        ua_frame[1] = A_SENDER;
        ua_frame[2] = C_UA;
        ua_frame[3] = A_SENDER ^ C_UA;
        ua_frame[4] = FLAG;

            write_to_port(fd, ua_frame, 5);
        printf("Sent UA frame : ");
        print_frame(ua_frame, 5);
    }

    if (type == UA_SENDER){
        char ua_frame[5];
        ua_frame[0] = FLAG;
        ua_frame[1] = A_RECEIVER;
        ua_frame[2] = C_UA;
        ua_frame[3] = A_RECEIVER ^ C_UA;
        ua_frame[4] = FLAG;

        write_to_port(fd, ua_frame, 5);
        printf("Sent UA frame : ");
        print_frame(ua_frame, 5);
    }

    if(type == DISC_RECEIVER){
        char disc_frame[5];
        disc_frame[0] = FLAG;
        disc_frame[1] = A_RECEIVER;
        disc_frame[2] = C_DISC;
        disc_frame[3] = A_RECEIVER ^ C_DISC;
        disc_frame[4] = FLAG;

        write_to_port(fd, disc_frame, 5);
        printf("Sent DISC frame : ");
        print_frame(disc_frame, 5);
    }

    if (type == DISC_SENDER){
        char disc_frame[5];
        disc_frame[0] = FLAG;
        disc_frame[1] = A_SENDER;
        disc_frame[2] = C_DISC;
        disc_frame[3] = A_SENDER ^ C_DISC;
        disc_frame[4] = FLAG;

        write_to_port(fd, disc_frame, 5);
        printf("Sent DISC frame : ");
        print_frame(disc_frame, 5);
    }
}

unsigned char calculate_bcc2(char* data, size_t size) {
    unsigned char result = data[0];

    for(int i = 1; i < size; ++i)
        result ^= data[i];

    return result;
}

int stuff_data(char* data, size_t size, char* stuffed) {
```

```
    int i = 1, j = 1;
    stuffed[0] = data[0];

    for(; i < size - 1; ++i) {

        if(data[i] == FLAG) {
            stuffed[j] = ESC;
            stuffed[++j] = SEQ_1;
        } else if(data[i] == ESC) {
            stuffed[j] = ESC;
            stuffed[++j] = SEQ_2;
        } else {
            stuffed[j] = data[i];
        }
        j++;
    }
    stuffed[j] = data[size - 1];

    return j+1;
}

void send_rr(int fd, char c) {
    char rr[5];
    rr[0] = FLAG;
    rr[1] = A_SENDER;

    if(c == C_INFO1)
        rr[2] = C_RR_1;
    else if(c == C_INFO2)
        rr[2] = C_RR_2;

    rr[3] = A_SENDER ^ rr[2];
    rr[4] = FLAG;

    printf("Sent RR: ");
    print_frame(rr, 5);

    write_to_port(fd, rr, 5);
}

void send_rej(int fd, char c) {
    char rej[5];
    rej[0] = FLAG;
    rej[1] = A_SENDER;

    if(c == C_INFO1)
        rej[2] = C_REJ_1;
    else if(c == C_INFO2)
        rej[2] = C_REJ_2;

    rej[3] = A_SENDER ^ rej[2];
    rej[4] = FLAG;

    printf("Sent REJ: ");
    print_frame(rej, 5);

    write_to_port(fd, rej, 5);
}
```

common.h

```
#ifndef _COMMON_H
#define _COMMON_H

typedef enum {
    UA_SENDER,
    UA_RECEIVER,
    SET,
    DISC_SENDER,
    DISC_RECEIVER,
    RR,
    REJ
} frameType;

#define MAX_CHUNK_SIZE 400
```

```
#define MAX_PACKET_SIZE MAX_CHUNK_SIZE + 4
#define MAX_FRAME_SIZE MAX_PACKET_SIZE + 6

void write_to_port(int fd, char* data, size_t s);
void send_frame(int fd, frameType type);

void print_frame(char* frame, size_t s);

unsigned char calculate_bcc2(char* data, size_t size);
int stuff_data(char* data, size_t size, char* stuffed);

void send_rr(int fd, char c);
void send_rej(int fd, char c);

#endif
```

message_macros.h

```
#ifndef MESSAGE_MACROS_H
#define MESSAGE_MACROS_H


/*  Supervisao e nao numeradas  */
/* [F,A,C,BCC1(A^C),F] */

#define FLAG                    0x7E

#define A_RECEIVER              0x01
#define A_SENDER        0x03

#define C_SET           0x03
#define C_DISC          0x0B
#define C_UA            0x07
#define C_RR_1          0x05
#define C_RR_2          0x85
#define C_REJ_1         0x01
#define C_REJ_2         0x81

#define C_INFO1         0x40
#define C_INFO2         0x00

#define ESC             0x7D
#define SEQ_1           0x5E
#define SEQ_2           0x5D

#endif
```

read.c

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

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include "message_macros.h"
#include "common.h"
#include "state_machine.h"
#include "read.h"

#define BAUDRATE B38400
#define _POSIX_SOURCE 1 /* POSIX compliant source */
#define FALSE 0
#define TRUE 1

static volatile int STOP = FALSE;

#define MAX_FRAME_TRIES 3
```

```c
#define FRAME_TIMEOUT 3


static int disc_tries = 0;
static int alarm_flag = 0;
static int retry_flag = 0;

unsigned int timeout;
unsigned int numTransmissions;

int read_frame_reader(int fd, char* data, frame_type frame_type){
        char result;
        STATE st;
        int stuffed = 0;
        int i = 0;
        unsigned char c;

        printf("Trying to read frame:\n");

        while(STOP == FALSE) {

                read(fd, &result, 1);
                if(retry_flag == 1){
                        return -1;
                }

                if(result == ESC) {
                        stuffed = 1;
                        continue;
                }
                if(stuffed) {
                        if(result == SEQ_1)
                                data[i] = FLAG;
                        else if(result == SEQ_2)
                                data[i] = ESC;
                        else {
                                send_rej(fd, c);
                                printf("REJ\n");
                                return -2;
                        }
                        stuffed = 0;
                } else
                        data[i] = result;
                i++;

                st = machine(result, RECEIVER_M, frame_type);

                if(st == STOP_ST || st == STOP_INFO) {
                        STOP = TRUE;
                } else if(st == C_RCV) {
                        c = result;
                } else if(st == START) {
                        i = 0;
                }
        }
        STOP = FALSE;

        if(st == STOP_ST){
                return 5; //size of the SET buffers
        }

        printf("RECEIVED INFO %s : %d\n", data, i);


        int j = 5;
        unsigned char bcc2_check = data[j-1];

        for(; j < i - 2; ++j)
        bcc2_check ^= data[j];

        if((unsigned char)(data[i - 2]) != bcc2_check) {
                send_rej(fd, c);
                return -2;
        } else {
                send_rr(fd, c);
        }

        int k = 4;
```

```
        for(; k < i - 2; ++k){
                data[k - 4] = data[k];
        }

        // print_frame(*data, i);

        return k - 4;
}

void read_set(int fd){
        char* frame = (char*)malloc(sizeof(char) * MAX_FRAME_SIZE);

        while(1){
                read_frame_reader(fd, frame, COMMAND);
                if(frame[2] == C_SET){
                        printf("Received SET frame\n");
                        send_frame(fd, UA_RECEIVER);
                        break;
                }
        }
        free(frame);
}

void read_disc(int fd){
        char* frame = (char*)malloc(sizeof(char) * MAX_FRAME_SIZE);

        while(1){
                read_frame_reader(fd, frame, COMMAND);
                if(frame[2] == C_DISC){
                        printf("Received DISC frame\n");
                        break;
                }
        }
        free(frame);
}

int read_info(int fd, char* buffer){
        int read_size;

        do {
                if((read_size = read_frame_reader(fd, buffer, COMMAND)) == -1)
                        return -1;
        } while(read_size == -2);
        return read_size;
}

void disc_alarm_receiver(){
        struct sigaction a;
        a.sa_handler = sigalarm_disc_handler_reader;
        a.sa_flags = 0;
        sigemptyset(&a.sa_mask);
        sigaction(SIGALRM, &a, NULL);
        printf("DISC Alarm handler set\n");
}

int send_disc_receiver(int fd){

    disc_alarm_receiver();

        alarm(FRAME_TIMEOUT);
    send_frame(fd, DISC_RECEIVER);

        char* ua_frame = (char*)malloc(sizeof(char) * MAX_FRAME_SIZE);

        while(alarm_flag == 0){
                read_frame_reader(fd, ua_frame, RESPONSE);
                if(retry_flag == 1){
                        send_frame(fd, DISC_RECEIVER);
                        retry_flag = 0;
                        continue;
                }
                if(ua_frame[2] == C_UA){
                        printf("Received UA\n");
                        alarm(0);
                        free(ua_frame);
                        return 0;
                } else {
                        printf("Wasn't UA\n");
```

```
                }
        }
        alarm_flag = 0;
        free(ua_frame);
        return -1;
}

void sigalarm_disc_handler_reader(int sig){
    if (disc_tries < MAX_FRAME_TRIES){
        printf("Alarm timeout\n");
        disc_tries++;
                retry_flag = 1;
        // send_frame(fd, DISC_RECEIVER);
        alarm(FRAME_TIMEOUT);
    } else {
        perror("DISC max tries reached\n");
        alarm_flag = 1;
    }
}
```

read.h

```
#ifndef READ_H
#define READ_H

#include "state_machine.h"

int read_frame_reader(int fd, char* data, frame_type frame_type);

void read_set(int fd);
void read_disc(int fd);
int read_info(int fd, char* buffer);

int send_disc_receiver(int fd);
void disc_alarm_receiver();
void sigalarm_disc_handler_reader(int sig);

#endif
```

state_machine.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include "common.h"
#include "state_machine.h"
#include "message_macros.h"

STATE receiver_state = START;
STATE sender_state = START;

STATE machine(unsigned char input, machine_type type, frame_type frame_type){

    STATE st;

    if(type == RECEIVER_M)
        st = receiver_state;
    else
        st = sender_state;
    // printf("STATE: %d\n", st);

    static unsigned char c;

    if(st == STOP_ST || st == STOP_INFO){
        st = START;
    }

    switch (st){
        case START:
            if(input == FLAG)
                st = FLAG_RCV;
            break;

        case FLAG_RCV:
            if(frame_type == COMMAND){
```

```
                if((type == RECEIVER_M && input == A_SENDER) ||
                    (type == SENDER_M && input == A_RECEIVER))
                    st = A_RCV;
                else if(input != FLAG)
                    st = START;
            } else {
                if((type == RECEIVER_M && input == A_RECEIVER) ||
                    (type == SENDER_M && input == A_SENDER))
                    st = A_RCV;
                else if(input != FLAG)
                    st = START;
            }
            break;

        case A_RCV:
            if(type == RECEIVER_M && frame_type == COMMAND && ((unsigned char)input == C_RR_1 ||
(unsigned char)input == C_RR_2 ||
                (unsigned char)input == C_REJ_1 || (unsigned char)input == C_REJ_2)) {
                st = START;
            } else if(input == FLAG)
                st = FLAG_RCV;
            else {
                st = C_RCV;
                c = input;
            }
            break;

        case C_RCV:
            if(input == FLAG)
                st = FLAG_RCV;
            else if(frame_type == COMMAND){
                if((type == RECEIVER_M && input == (A_SENDER ^ c)) ||
                (type == SENDER_M && input == (A_RECEIVER ^ c)))
                    st = BCC_OK;
                else
                    st = START;

            } else if(frame_type == RESPONSE){
                if((type == RECEIVER_M && input == (A_RECEIVER ^ c)) ||
                (type == SENDER_M && input == (A_SENDER ^ c)))
                    st = BCC_OK;
                else
                    st = START;
            }
            break;

        case BCC_OK:
            if(input == FLAG)
                st = STOP_ST;
            else {
                st = INFO;
            }
            break;

        case INFO:
            if(input == FLAG)
                st = STOP_INFO;
            break;

        case STOP_ST:
            break;

        case STOP_INFO:
            break;

        default:
            break;
    }

    if(type == RECEIVER_M)
        receiver_state = st;
    else
        sender_state = st;

    return st;
}

void reset_state(machine_type type){
```

```
    if(type == RECEIVER_M)
        receiver_state = START;
    else
        sender_state = START;
}
```

state_machine.h

```
#ifndef _STATE_MACHINE_H
#define _STATE_MACHINE_H

typedef enum {
    START       = 0,
    FLAG_RCV    = 1,
    A_RCV       = 2,
    C_RCV       = 3,
    BCC_OK      = 4,
    STOP_ST     = 5,
    STOP_INFO   = 6,
    INFO        = 7
} STATE;

typedef enum {
    RECEIVER_M,
    SENDER_M
} machine_type;

typedef enum {
    COMMAND,
    RESPONSE
} frame_type;

STATE machine(unsigned char input, machine_type type, frame_type frame_type);

void reset_state(machine_type type);

#endif
```

write.c

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

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include <errno.h>
#include "message_macros.h"
#include "common.h"
#include "state_machine.h"
#include "write.h"

#define BAUDRATE B38400
#define MODEMDEVICE "/dev/ttyS11"
#define _POSIX_SOURCE 1 /* POSIX compliant source */
#define FALSE 0
#define TRUE 1

#define MAX_FRAME_TRIES         3
#define FRAME_TIMEOUT           3

static volatile int STOP = FALSE;

static int set_tries = 0;
static int disc_tries = 0;
static int info_tries = 0;
static int alarm_flag = 0;
static int retry_flag = 0;

static char* lastSent;
static size_t last_sent_size = 0;
```

```
void read_frame_writer(int fd, char* out, frame_type frame_type){

        printf("--Trying to read frame--\n");

        unsigned char result;
        STATE st;

        while (STOP == FALSE){
                read(fd, &result, 1);

                if(alarm_flag == 1){
                        reset_state(SENDER_M);
                        return;
                }
                if(retry_flag == 1){
                        return;
                }
                if((st = machine(result, SENDER_M, frame_type)) > 0)
                        out[st - 1] = result;

                if(st == STOP_ST)
                        STOP = TRUE;
        }
        STOP = FALSE;

        printf("--Received frame--\n");
}

void send_info_frame(int fd, char* data, size_t size, int resend) {

    static int c = 0;

        if(resend == TRUE){
                write_to_port(fd, lastSent, last_sent_size);
                return;
        }

    char* frame = (char*)malloc(sizeof(char) * (size + 6));

    frame[0] = FLAG;
    frame[1] = A_SENDER;

    if(c % 2 == 0)
        frame[2] = C_INFO1;
    else frame[2] = C_INFO2;

    frame[3] = A_SENDER ^ frame[2];

    int i = 0;
    for(; i < size; ++i) {
        frame[i + 4] = data[i];
    }

    frame[i + 4] = calculate_bcc2(data, size);
    frame[i + 5] = FLAG;

    char* stuffed_frame = (char*)malloc(sizeof(char) * 2 * (size + 6));

    int frame_size = stuff_data(frame, i + 6, stuffed_frame);

    c++;

        lastSent = (char*)malloc(sizeof(char) * frame_size);
        for(int k = 0; k < frame_size; k++){
                lastSent[k] = stuffed_frame[k];
        }
        last_sent_size = frame_size;

    write_to_port(fd, stuffed_frame, frame_size);
    printf("Sent INFO frame : %s : %d\n", stuffed_frame, frame_size);
        free(stuffed_frame);
        free(frame);
}

void sigalarm_set_handler_writer(int sig){
        if(set_tries < MAX_FRAME_TRIES){
                printf("Alarm timeout\n");
```

```
                set_tries++;
                retry_flag = 1;
                // send_frame(fd, SET);
                alarm(FRAME_TIMEOUT);
        } else {
                perror("SET max tries reached\n");
                alarm_flag = 1;
        }
}

void sigalarm_disc_handler_writer(int sig){
    if (disc_tries < MAX_FRAME_TRIES){
        printf("Alarm timeout\n");
        disc_tries++;
                retry_flag = 1;
        // send_frame(fd, DISC_SENDER);
        alarm(FRAME_TIMEOUT);
    } else {
        perror("DISC max tries reached\n");
        alarm_flag = 1;
    }
}

void sigalarm_info_handler_writer(int sig){
    if (info_tries < MAX_FRAME_TRIES){
        printf("Alarm timeout\n");
        info_tries++;
                retry_flag = 1;
        // send_info_frame(fd, lastSent, last_sent_size, TRUE);
        alarm(FRAME_TIMEOUT);
    } else {
        perror("INFO max tries reached\n");
        alarm_flag = 1;
    }
}

void set_alarm(){
        struct sigaction a;
        a.sa_handler = sigalarm_set_handler_writer;
        a.sa_flags = 0;
        sigemptyset(&a.sa_mask);
        sigaction(SIGALRM, &a, NULL);
        printf("SET Alarm handler set\n");
}

void disc_alarm_writer(){
        struct sigaction a;
        a.sa_handler = sigalarm_disc_handler_writer;
        a.sa_flags = 0;
        sigemptyset(&a.sa_mask);
        sigaction(SIGALRM, &a, NULL);
        printf("DISC Alarm handler set\n");
}

void info_alarm(){
        struct sigaction a;
        a.sa_handler = sigalarm_info_handler_writer;
        a.sa_flags = 0;
        sigemptyset(&a.sa_mask);
        sigaction(SIGALRM, &a, NULL);
        printf("INFO Alarm handler set\n");
}

int send_set(int fd){

        set_alarm();

        alarm(FRAME_TIMEOUT);

        char ua_frame[255];
        //First send
        send_frame(fd, SET);

        while(alarm_flag == 0){
                read_frame_writer(fd, ua_frame, RESPONSE);
                if(retry_flag == 1){
                        send_frame(fd, SET);
                        retry_flag = 0;
```

```
                        continue;
                }
                if(ua_frame[2] == C_UA){
                        printf("Received UA\n");
                        set_tries = 0;
                        alarm(0);
                        return 0;
                } else {
                        printf("Wasn't UA\n");
                }
        }
        alarm_flag = 0;
        return -1;
}

int send_disc_sender(int fd){
        char disc_frame[255];

    disc_alarm_writer();

        alarm(FRAME_TIMEOUT);
    send_frame(fd, DISC_SENDER);

        while(alarm_flag == 0){
                read_frame_writer(fd, disc_frame, COMMAND);
                if(retry_flag == 1){
                        send_frame(fd, DISC_SENDER);
                        retry_flag = 0;
                        continue;
                }
                if(disc_frame[2] == C_DISC){
                        printf("Received DISC\n");
                        disc_tries = 0;
                        alarm(0);
                        send_frame(fd, UA_SENDER);
                        return 0;
                } else {
                        printf("Wasn't DISC\n");
                }
        }
        alarm_flag = 0;
        return -1;
}

int send_info(int fd, char* data, int length){

        info_tries = 0;

        info_alarm();

        alarm(FRAME_TIMEOUT);

        char response[255];
        int resend = FALSE;
        while(alarm_flag == 0){
                if(retry_flag){
                        send_info_frame(fd, lastSent, last_sent_size, TRUE);
                        retry_flag = 0;
                } else {
                        send_info_frame(fd, data, length, resend);
                }
                read_frame_writer(fd, response, RESPONSE);

                if(retry_flag == 1 || alarm_flag == 1){
                        continue;
                }

                print_frame(response, 5);

                if((unsigned char)(response[2]) == C_RR_1 || (unsigned char)(response[2]) == C_RR_2) {
                        printf("RR received\n");
                        resend = FALSE;
                        alarm(0);
                        free(lastSent);
                        return length;
                } else if((unsigned char)(response[2]) == C_REJ_1 || (unsigned char)(response[2]) ==
C_REJ_2) {
                        printf("REJ received\n");
```

```
                        resend = TRUE;
                }
        }
        free(lastSent);
        alarm_flag = 1;
        return -1;
}
```

write.h

```
#ifndef WRITE_H
#define WRITE_H

#include <stdlib.h>
#include "state_machine.h"

void read_frame_writer(int fd, char* out, frame_type frame_type);
void send_info_frame(int fd, char* data, size_t size, int resend);

void sigalarm_set_handler_writer(int sig);
void sigalarm_disc_handler_writer(int sig);
void sigalarm_info_handler_writer(int sig);
void set_alarm();
void info_alarm();
void disc_alarm_writer();

int send_set(int fd);
int send_disc_sender(int fd);
int send_info(int fd, char* data, int length);

#endif
```

Anexo II (Dados de eficiência)

NOTA: Todos os testes foram realizados utilizando um ficheiro de 87744 bits.

Eficiência em função do tamanho dos pacotes de dados:
Baudrate: 38400

Packet Size	t	R(bits/s)	S(R/C)	S(média)
30	5,353764	16389,217007	0,4268025262	0,4267891739
	5,354099	16388,191552	0,4267758217	
60	4,623763	18976,751187	0,4941862288	0,4941687018
	4,624091	18975,405112	0,4941511748	
90	4,377316	20045,160094	0,5220093774	0,5218951232
	4,379233	20036,385367	0,5217808689	
120	4,258503	20604,423667	0,536573533	0,5366388802
	4,257466	20609,44233	0,5367042274	
200	4,10977	21350,099884	0,5559921845	0,5559688497
	4,110115	21348,307772	0,5559455149	
400	4,001873	21925,733275	0,5709826374	0,5709839215
	4,001855	21925,831895	0,5709852056	
1000	3,934035	22303,817836	0,5808285895	0,5808303612
	3,934011	22303,953904	0,5808321329	
5000	3,902067	22486,54367	0,5855870747	0,5855868496
	3,90207	22486,526382	0,5855866245	

Eficiência em função da *baudrate* usada (para pacotes de 250 bytes):

Baudrate	t	R(bits/s)	S(R/C)	S(média)
1200	99,352798	883,15580201	0,7359631683	0,7375542389
	98,925068	886,97437135	0,7391453095	
1800	66,285525	1323,727918	0,7354043989	0,7354058855
	66,285257	1323,73327	0,7354073722	
2400	49,966586	1756,0535355	0,7316889731	0,7316900348
	49,966441	1756,0586314	0,7316910964	
4800	25,486522	3442,7608443	0,7172418426	0,7172436015
	25,486397	3442,7777296	0,7172453603	
9600	13,246763	6623,8068878	0,6899798841	0,6899801706
	13,246752	6623,8123881	0,6899804571	
19200	7,127515	12310,601942	0,6411771845	0,6411323467
	7,128512	12308,88017	0,6410875089	
38400	4,067198	21573,574731	0,561811842	0,5618282116
	4,066961	21574,83192	0,5618445812	

Eficiência em função da probabilidade de erro do BCC1 e BCC2
(para pacotes de 100 bytes e *baudrate* 38400):

Erro BCC(%)	t	R(bits/s)	S(R/C)	S(média)
0	4,301534	20398,304419	0,5312058442	0,5312187498
	4,301325	20399,295566	0,5312316554	
2	7,393176	11868,241741	0,3090687953	0,4183115292
	4,331308	20258,083701	0,5275542631	
4	13,511444	6494,0505249	0,1691158991	0,1529932069
	16,694611	5255,8277638	0,1368705147	
6	22,693249	3866,5243571	0,1006907385	0,089997004
	28,81344	3045,2455521	0,0793032696	
8	34,784515	2522,5017511	0,0656901498	0,0687913721
	31,783524	2760,6756255	0,0718925944	
10	32,077832	2735,3469524	0,0712329936	0,0682079479
	35,055205	2503,0234454	0,0651829022	

Eficiência em função do tempo de propagação (para pacotes de 100 bytes e *baudrate* 38400):

Tprop(ms)	t	R(bits/s)	S(R/C)	S(média)
5	5,022241	17471,085119	0,454976175	0,4549576957
	5,022649	17469,665907	0,4549392163	
10	5,581657	15720,063057	0,4093766421	0,4093723883
	5,581773	15719,736363	0,4093681345	
20	6,700732	13094,688759	0,3410075198	0,3409335562
	6,70364	13089,00836	0,3408595927	
40	8,94628	9807,8754521	0,2554134232	0,2554591619
	8,943077	9811,3881833	0,2555049006	
80	13,422195	6537,2318015	0,1702404115	0,170245111
	13,421454	6537,5927228	0,1702498105	
160	22,38176	3920,33513	0,1020920607	0,1020921314
	22,381729	3920,3405599	0,1020922021	