



RCOM

FEUP-MIEIC

---

## Trabalho Laboratorial #2

---

*Autor:*

Mariana Costa

Pedro Silva

Tiago Castro

*Numero de estudante:*

up201604414

up201604470

up201606186

# Contents

<b>1</b>	<b>Sumário</b>	<b>1</b>
<b>2</b>	<b>Introdução</b>	<b>1</b>
<b>3</b>	<b>Arquitetura</b>	<b>1</b>
3.1	Camada de Aplicação . . . . .	2
3.2	Camada de Ligação de Dados . . . . .	2
<b>4</b>	<b>Estrutura do código</b>	<b>2</b>
4.1	Camada da Aplicação . . . . .	2
4.2	Camada de Ligação de Dados . . . . .	3
<b>5</b>	<b>Casos de Uso Principais</b>	<b>4</b>
<b>6</b>	<b>Protocolo de Ligação Lógica</b>	<b>4</b>
6.1	Estratégias de Implementação . . . . .	5
<b>7</b>	<b>Protocolo de Aplicação</b>	<b>6</b>
7.1	Estratégias de Implementação . . . . .	7
<b>8</b>	<b>Validação</b>	<b>8</b>
<b>9</b>	<b>Eficiência do protocolo de ligação de dados</b>	<b>8</b>
<b>10</b>	<b>Conclusões</b>	<b>8</b>
<b>11</b>	<b>Anexos</b>	<b>9</b>
11.1	Anexo I . . . . .	9
11.2	Anexo II . . . . .	10
11.3	Anexo III . . . . .	11
11.4	Anexo IV . . . . .	12

## 1 Sumário

Para o 1º trabalho laboratorial foi-nos proposta a implementação de um protocolo de ligação de dados. Este protocolo deve atuar através da implementação de duas camadas, a Camada de Aplicação (Application Layer) e a Camada de Ligação de Dados (Data Link Layer).

O objetivo final do projeto é conseguir transferir um ficheiro entre dois computadores, através de uma porta de serie assíncrona, onde ambos conhecem o protocolo e seguem-no de modo a evitar erros de bits (interferência) e corte total da comunicação.

## 2 Introdução

O objetivo do projeto realizado é permitir a transferência de ficheiros entre dois computadores através de um protocolo de ligação de dados. Este protocolo foi devidamente especificado no ‘Guião do trabalho’, bem como no decorrer das aulas práticas. Este relatório tem como objetivo principal explicar o funcionamento do protocolo, a sua lógica e estruturação. Todo o código encontra-se no Anexo IV.

- Arquitetura – Blocos funcionais e interfaces;
- Estrutura do código – APIs, principais estruturas de dados, principais funções e relação com arquitetura;
- Casos de uso principais – Casos de uso e sequência de funções usadas;
- Protocolo de ligação lógica – Principais aspetos funcionais, descrição estratégica de implementação;
- Protocolo de aplicação – Principais aspetos funcionais, descrição estratégica de implementação;
- Validação – Métodos de validação da implementação do protocolo;
- Eficiência do protocolo de ligação de dados – Estatística da eficiência do protocolo. Caracterização teórica de um protocolo Stop-and-Wait e comparação com o usado;
- Conclusão – Síntese da informação apresentada no relatório e reflexão dos objetivos alcançados.

## 3 Arquitetura

O trabalho está fragmentado em duas camadas (layers) com funções muito específicas, sendo que sua diferença reside maioritariamente nas responsabilidades que tem de acordo com o seu grau de abstração.

### 3.1 Camada de Aplicação

É a camada de mais alto nível que envia, através do link layer, fragmentos do ficheiro escolhido pelo utilizador para que o recetor possa receber esses mesmo fragmentos também do link layer e reconstruir o ficheiro.

É considerada dependente da Camada de Ligação de Dados porque a sua função é servir esta com a informação a ser enviada.

### 3.2 Camada de Ligação de Dados

Esta camada garante a comunicação sem erros entre os dois computadores, a um nível de abstração muito baixa, visto que implementa, como referido anteriormente, as funções usadas pela Camada de Aplicação como o `lread` e o `llwrite`.

No entanto, é considerada independente da camada superior pois ela não interpreta os valores passados, apenas os trata como válidos e garante que eles cheguem ao outro computador exatamente como pedido. Para além disso também é responsável pela gestão a um mais baixo nível da porta de série, tanto a abertura como as propriedades da mesma.

## 4 Estrutura do código

### 4.1 Camada da Aplicação

#### Funções Comuns

Em ambos os casos, são usadas funções, implementadas na Camada de Ligação de Dados, que permitem a comunicação entre os dois computadores.

- **llopen** – Estabelece ligação pela porta de série e verifica se a ligação é feita com sucesso;
- **llclose** – Fecha a porta de série e verifica se o mesmo é feito com sucesso;
- **llwrite** – Escreve na porta de série;
- **lread** – Lê da porta de série.

#### Transmissor

```
1 unsigned char * openFile(unsigned char * file, int * fileSize);  
2 int sendFile(int fd, unsigned char * fileName, int fileNameSize);
```

A função `openFile` recebe o nome do ficheiro e o seu tamanho e abre-o, guarda-o num array de bytes e retorna-o.

A função `sendFile` recebe o file descriptor, o descritor da porta série, o conteúdo do file e o seu nome. Tem como objetivo repartir e enviar o ficheiro em questão para o link layer.

### Recetor

```
1 struct FileInfo{
2     unsigned int size;
3     unsigned char* name;
4     unsigned char* content;
5 };
```

A struct `FileInfo` tem como objetivo gerir as informações do ficheiro que vai receber, o seu tamanho, nome e conteúdo.

```
1 int getFileInfo(unsigned char* start);
```

A função `getFileInfo` preenche `size` e `name` de `FileInfo` com a trama que contém essa informação.

```
1 void readContent(int fd, unsigned char* start, unsigned int startSize);
```

A função `readContent` preenche `content` de `FileInfo`.

```
1 void createFile();
```

A função `createFile` cria um ficheiro com os valores guardados na struct `FileInfo`.

## 4.2 Camada de Ligação de Dados

### Funções em Comum

```
1 void writeControlMessage(int fd, unsigned char control);
```

A função `writeControlMessage` envia uma mensagem de controlo com o carater passado por argumento.

### Transmissor

```
1 int stopAndWaitControl(int fd, unsigned char control_sent, unsigned char
   control_expecting);
2 int stopAndWaitData(int fd, unsigned char * buffer, int length);
```

Seguindo o princípio de controlo de erro Stop-and-Wait foram criadas as funções acima declaradas. Estas funções vão enviando uma mensagem de controlo/informação ciclicamente até receber uma resposta de confirmação de envio ou sair por timeout (caso o recetor não responda ou demorar demasiado tempo).

A função `stopAndWaitControl` gere o envio de mensagens de controlo, enquanto que a `stopAndWaitData` gere o envio de mensagens de informação.

### Recetor

```
1 void readControlMessage(int fd, unsigned char control);  
2 unsigned int readPacket(int fd, unsigned char* buffer);
```

A função `readControlMessage` recebe e verifica se recebe uma mensagem de controlo em particular. O transmissor também utiliza esta função indiretamente durante as funções `stopAndWait`.

A função `readPacket` recebe e verifica uma mensagem de informação. Após a realização da operação de destuffing e a verificação do BCC2, retorna a trama de informação juntamente com o header. Caso o bcc2 não estiver correto, ou a trama não ser a esperada, envia o sinal REJ, esperando então o reenvio do packet. Em caso contrário, envia RR.

## 5 Casos de Uso Principais

Esta aplicação tem como principal caso de uso a chamada ao programa, que permite ao transmissor a escolha de que ficheiro pretende enviar ao recetor. Para além disso, tanto o transmissor como o recetor, devem explicitar qual a porta de série a ser utilizada.

Exemplo das chamadas em caso de envio de um ficheiro chamado `pinguim.gif`:

- Transmitter: `gcc transmitter.c -o transmitter -Wall -Wextra`
  - `./transmitter /dev/ttyS0 pinguim.gif`
- Receiver: `gcc receiver.c -o receiver -Wall -Wextra`
  - `./receiver /dev/ttyS0`

## 6 Protocolo de Ligação Lógica

A Camada da Ligação Lógica é responsável pelos aspetos funcionais seguintes:

- CONFIG - Configuração da porta de série;
- OPEN – Abertura da porta de série;
- CLOSE - Terminação da ligação através da porta de série;
- WRITE - Criação e envio de comandos e mensagens através da porta de série;
- READ - Receção de comandos e mensagens através da porta de série;

- CHECK - Verificação de erros durante transferência de dados;
- STUFF - Stuffing dos pacotes enviados pela Camada de Aplicação;
- DESTUFF - Destuffing dos pacotes enviados pela Camada de Aplicação.

## 6.1 Estratégias de Implementação

### llopen

Inicia a ligação da porta de série (OPEN) e altera as suas configurações (CONFIG) para que a leitura possa ser feita sem interrupções.

```
1 newtio.c_cc[VTIME] = 1; /* inter-character timer unused */
2 newtio.c_cc[VMIN] = 0; /* don't stop trying to read */
```

Após isso, o transmissor entra no ciclo com controlo de erro stopAndWait onde vai enviando (SEND) tramas de inicialização até receber a resposta esperada – UA (trama de reconhecimento).

Transmissor:

```
1 stopAndWaitControl(fd, CONTROLSET, CONTROLUA);
```

Recetor:

```
1 readControlMessage(fd, CONTROLSET);
2 writeControlMessage(fd, CONTROLUA);
```

Caso a resposta não chegue dentro de um tempo especificado como timeout, a mensagem volta a ser enviada até ultrapassar um valor pré-definido (CHECK). Caso houver algum problema na conexão a função deve retornar -1, terminando o processo.

### llclose

O transmissor entra num ciclo de stopAndWait no qual envia tramas de desconexão e espera uma confirmação do recetor para que possa enviar, finalmente, uma trama UA, de forma a que ambos os processos possam encerrar (CLOSE).

Transmissor:

```
1 stopAndWaitControl(fd, CONTROLDISC, CONTROLDISC);
2 sendControlMessage(fd, CONTROLUA);
3 tcsetattr(fd, TCSANOW, &oldtio);
```

Recetor:

```
1 writeControlMessage(fd, CONTROLDISC);
2 readControlMessage(fd, CONTROLUA);
3 tcsetattr(fd, TCSANOW, &oldtio);
```

## llwrite

Responsável pela escrita na porta de série (WRITE). Esta função é responsável pelo Stuffing do pacote recebido pela Camada de Aplicação, e de o encapsular de forma a criar uma trama I (de informação) (STUFF).

```
1 generateBCC2(buffer, length);
2 BCC2Stuffing(&BCC2, &stuffedBCC2Size);
3 packetStuffing(buffer, length, &finalPacketSize);
4 preparePacket(finalPacket, stuffedBCC2, &finalPacketSize, &stuffedBCC2Size);
```

## llread

Responsável pela leitura na porta de série (READ). Esta função lê informação da porta de série, verifica se houve erros na comunicação (através da paridade da informação) e envia confirmação de volta ao transmissor.

Caso a informação tenha erros, o recetor responderá com um comando REJ. Caso não haja erro, o llread faz destuff do pacote inicialmente criado pela Camada de Aplicação, tanto informação como o BCC (DESTUFF). Após o destuff, verifica se a informação enviada tem a mesma paridade do que o BCC, confirmando a validade da informação. Em caso positivo, é enviado uma trama RR (mesmo que o pacote seja repetivo).

```
1 readPacket(fd, buffer);
2 destuffing(buffer, packetSize);
3 checkBCC2(buffer, packetSize);
4 //In case of sucess
5 writeControlMessage(fd, RR.C_?); //Where '?' is the packet sign, either 0 or
  1
6 //In case of an error
7 writeControlMessage(fd, REJ.C_?); //Where '?' is the packet sign, either 0 or
  1
```

## 7 Protocolo de Aplicação

A Camada de Aplicação é responsável pelos aspetos funcionais seguintes:

- CREATE – Criação dos pacotes de dados e de controlo;
- INTERPRET - Interpretação dos pacotes de dados e de controlo;
- SCAN – Leitura do ficheiro a ser criado;
- GENERATE – Gera o ficheiro recebido;
- UNION - Junção das payloads para obter a informação do ficheiro.
- SPLIT - Fragmentação do conteúdo do ficheiro em vários pacotes



## 7.1 Estratégias de Implementação

### createFile

Cria o ficheiro utilizando o bloco de dados que foi preenchido ao longo da evolução do programa (GENERATE).

```
1 createFile();
```

### splitData

Divide o ficheiro em packets cujo tamanho pode ser modificado pelo utilizador (por default, é-lhe atribuído o valor de 255) (SPLIT).

```
1 currPacket = splitData(fileData, fileSize, &currPos, &currPacketSize);
```

### prepareDataPacketHeader

Anexa a um array de dados do ficheiro o header, contendo o campo de controlo (neste caso, terá o valor de 1, de forma a indicar que se trata de um packet de dados), o número de sequência e o número de octetos do campo de dados (CREATE).

```
1 currPacket = prepareDataPacketHeader(currPacket, fileSize, &currPacketSize, &numPackets);
```

### prepareAppControlPacket

Cria os packets de controlo que permitem indicar o início e fim da transmissão de informação relativa ao ficheiro em questão (CREATE).

```
1 unsigned char * startPacket = prepareAppControlPacket(APP_CONTROLSTART,
    fileSize, fileName, fileNameSize, &appControlPacketSize);
```

### openFile

Lê a info do ficheiro (SCAN) e retorna-a para divisão em pacotes e enviada (SPLIT).

```
1 unsigned char * fileData = openFile(fileName, &fileSize);
```

### readContent

Responsável por receber os packets do lread. Analisa cada um a ver se é o packet final (compara todos os bits de cada packet recebido com o packet de start que é passado por parametro) (INTERPRET) e remove o header caso não seja e adiciona-o ao bloco de dados que contem o conteúdo do ficheiro (UNION). Caso seja o packet final, termina o loop.

```
1 readContent(fd, start, startSize);
2 if(isEndPacket(start, startSize, packet, packetSize)) break;
3 packetSize = removeHeader(packet, packetSize);
4 memcpy(info.content + index, packet, packetSize);
```

### **getFileInfo**

Interpreta o conteúdo da trama start e analisa a sua autenticidade (INTERPRET).

```
1 getFileInfo(start);
```

## **8 Validação**

Após discussão com o docente, chegámos à conclusão de que a melhor avaliação da validação do protocolo e da comunicação seria por verificação visual do ficheiro transferido.

Na apresentação foi transferido o ficheiro de teste presente no moodle ‘pinguim.gif’ e foi constatado que o ficheiro criado no recetor era igual ao enviado pelo transmissor. Tanto o ficheiro controlo como o ficheiro gerado pelo programa encontram-se no Anexo III.

## **9 Eficiência do protocolo de ligação de dados**

De forma a avaliar quantitativamente a eficiência do protocolo utilizado foram realizados vários testes que permitiram a geração de gráficos e tabelas correspondentes, devidamente identificados nos Anexos I e II.

## **10 Conclusões**

O objetivo deste trabalho era compreender a importância de protocolos de dados para a comunicação correta e consistente entre vários computadores e consequentemente redes.

Um protocolo permite que vários computadores saibam como interpretar informação vinda de fora e saibam como formatar a informação que pretendem enviar para que seja bem recebida e interpretada.

A implementação do protocolo do trabalho laboratorial permitiu a aprendizagem da separação de vários níveis de abstração e a sua interindependência. Neste protocolo em particular, a camada da ligação de dados não interpreta o cabeçalho ou informação dado pela camada de aplicação, apenas se limitando a enviar/receber.

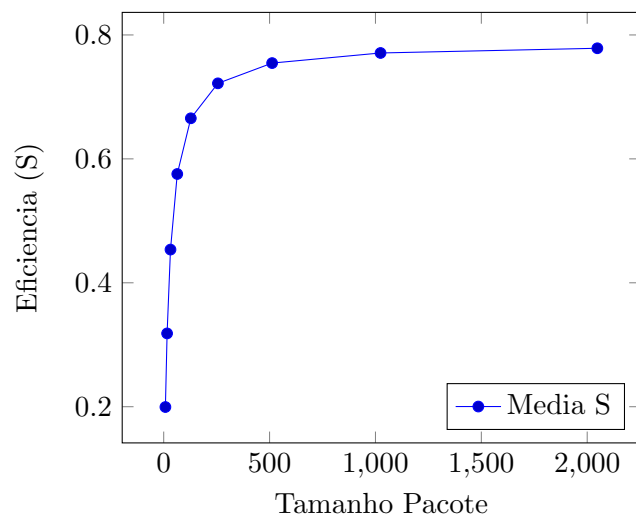
Concluindo, o projeto foi completado com sucesso, cumprindo-se todos os objetivos e consequente compreensão mais profunda do conceito de comunicação entre computadores e redes, algo essencial na criação de algo que todos nós usamos diariamente, a internet.

## 11 Anexos

### 11.1 Anexo I

Para um ficheiro com 87744 bits e baudrate a 38400

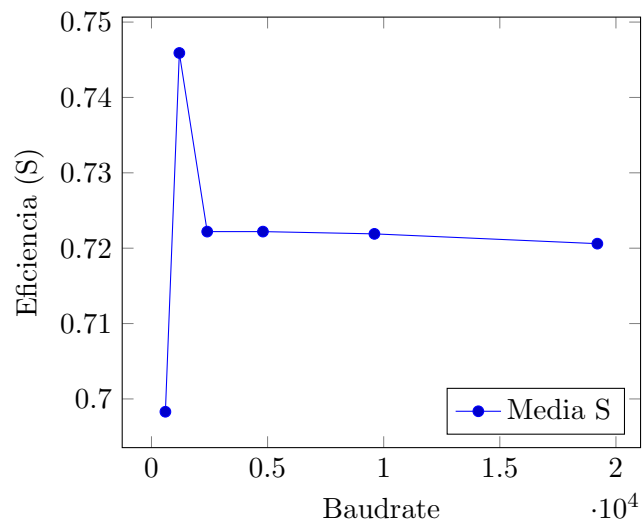
Tamanho Pacote	Time 1	Time 2	R 1	R 2	S 1	S 2	Media S
8	11.46	11.45	7,656.5	7,663.2	0.1994	0.1995	0.1995
16	7.181	7.175	12,218	12,229	0.3182	0.3183	0.3183
32	5.04	5.038	17,409	17,416	0.4536	0.4535	0.4536
64	3.971	3.968	22,096	22,112	0.5754	0.5758	0.5756
128	3.433	3.433	25,558	25,558	0.6655	0.6655	0.6655
256	3.165	3.164	27,723	27,723	0.7219	0.7219	0.7219
512	3.028	3.028	28,977	28,977	0.7546	0.7546	0.7546
1,024	2.964	2.964	29,603	29,603	0.7709	0.7709	0.7709
2,048	2.935	2.935	29,895	29,895	0.7785	0.7785	0.7785



## 11.2 Anexo II

Para um ficheiro com 87744 bits e packets com 256 bits

Baudrate	Time 1	Time 2	R 1	R 2	S 1	S 2	Media S
19,200	6.339	6.337	13,841	13,846	0.7201	0.7211	0.7206
9,600	12.66	12.66	6,930.8	6,930.8	0.7219	0.7219	0.7219
4,800	25.31	25.31	3,466.7	3,466.7	0.7222	0.7222	0.7222
2,400	50.62	50.62	1,733.4	1,733.4	0.7222	0.7222	0.7222
1,200	98.03	98.03	895.07	895.07	0.7461	0.7458	0.7459
600	209.3	209.4	419.22	419.22	0.6983	0.6983	0.6983



### 11.3 Anexo III

Imagem controlo do moodle

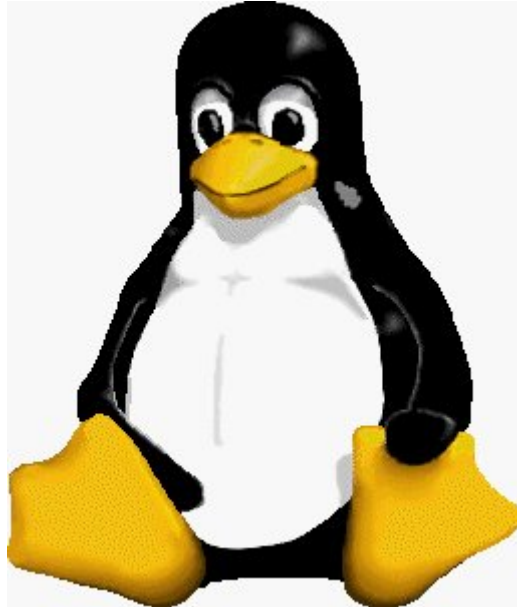
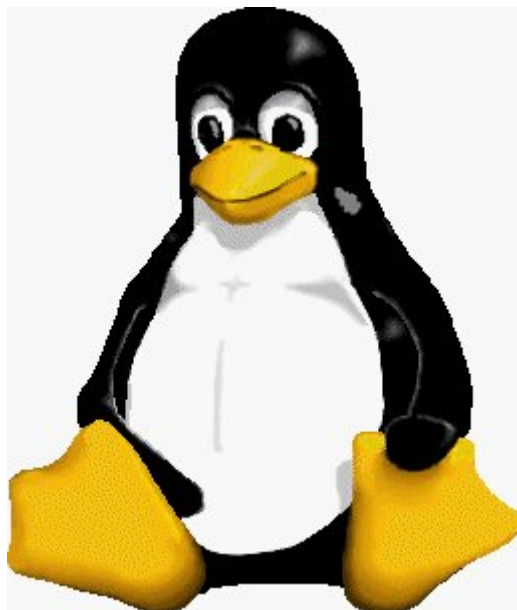


Imagem enviada e reenviada usando o programa desenvolvido



## 11.4 Anexo IV

### receiver.h

```
1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <fcntl.h>
4 #include <termios.h>
5 #include <unistd.h>
6 #include <stdio.h>
7 #include <stdlib.h>
8 #include <strings.h>
9 #include <string.h>
10 #include <signal.h>
11
12 /* baudrate settings are defined in <asm/termbits.h>, which is
13 included by <termios.h> */
14 #define BAUDRATE          B38400
15 #define _POSIX_SOURCE      1 /* POSIX compliant source */
16 #define FALSE              0
17 #define TRUE               1
18
19 #define TIMEOUT           3
20
21 #define FLAG               0x7E
22 #define C_SET              0x03
23 #define C_UA               0x07
24 #define A                  0x03
25 #define C_DISC             0x0B
26 #define C_0                0x00
27 #define C_1                0x40
28 #define ESC                0x7D
29 #define ESC_FLAG           0x5E
30 #define ESC_ESC            0x5D
31 #define RR_C_0             0x05
32 #define RR_C_1             0x85
33 #define REJ_C_0            0x01
34 #define REJ_C_1            0x81
35 #define DATA              0x01
36 #define START              0x02
37 #define END                0x03
38 #define SIZE               0x00
39 #define NAME               0x01
40
41 struct FileInfo{
42     unsigned int size;
43     unsigned char* name;
44     unsigned char* content;
45 };
46
47 int llOpen(int fd);
```

```
48 int llClose(int fd);
49 void readControlMessage(int fd, unsigned char control);
50 void writeControlMessage(int fd, unsigned char control);
51 unsigned int llread(int fd, unsigned char* buffer);
52 int destuffing(unsigned char* buffer, unsigned int packetSize);
53 int checkBCC2(unsigned char* buffer, unsigned int packetSize);
54 unsigned int readPacket(int fd, unsigned char* buffer);
55 int getFileInfo(unsigned char* start);
56 void readContent(int fd, unsigned char* start, unsigned int startSize);
57 unsigned int isEndPacket(unsigned char* start, unsigned int startSize,
    unsigned char* end, unsigned int endSize);
58 unsigned int removeHeader(unsigned char* packet, unsigned int size);
59 void createFile();
```

**receiver.c**

```
1  /*Non-Canonical Input Processing*/
2  #include "receiver.h"
3
4  struct termios oldtio, newtio;
5  int packet;
6  int expected = 0;
7  struct FileInfo info;
8  unsigned char* changed;
9
10 int numPackets = 0;
11 int numRR = 0;
12 int numREJ = 0;
13
14 int main(int argc, unsigned char** argv){
15
16     int fd;
17
18     if ( (argc < 2) ||
19         ((strcmp("/dev/ttyS0", argv[1])!=0) &&
20          (strcmp("/dev/ttyS1", argv[1])!=0) )) {
21         printf("Usage:\tnserial SerialPort\n\tex: nserial /dev/ttyS1\n");
22         exit(1);
23     }
24     /*
25      Open serial port device for reading and writing and not as controlling
26      tty
27      because we don't want to get killed if linenoise sends CTRL-C.
28     */
29     fd = open(argv[1], ORDWR | O_NOCTTY);
30     if (fd < 0) {perror(argv[2]); exit(-1); }
31     llopen(fd);
32
33     unsigned char* start = (unsigned char*)malloc(sizeof(unsigned char));
34     if(start == NULL){
35         printf("Tried to malloc, out of memory\n");
36         exit(-1);
37     }
38
39     unsigned int startSize = lread(fd, start);
40     start = changed;
41     if(getFileInfo(start) == -1){
42         printf("File Size and File Name not in the correct order, first size,
43             then name\n");
44         return -1;
45     }
46
47     info.content = (unsigned char*)malloc(info.size * sizeof(unsigned char));
48     if(info.content == NULL){
49         printf("Tried to malloc, out of memory\n");
```



```
49     exit(-1);
50 }
51
52 readContent(fd, start, startSize);
53
54 printf("Number of packets read: %i\nNumber of packets rejected: %i\nNumber
    of packets accepted: %i\n", numPackets, numREJ, numRR);
55
56 createFile();
57
58 free(start);
59 free(info.name);
60 free(info.content);
61 readControlMessage(fd, C_DISC);
62
63 llClose(fd);
64 return 0;
65 }
66
67 int llOpen(int fd){
68     if (tcgetattr(fd, &oldtio) == -1) { /* save current port settings */
69         perror("tcgetattr");
70         exit(-1);
71     }
72
73     bzero(&newtio, sizeof(newtio));
74     newtio.c_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;
75     newtio.c_iflag = IGNPAR;
76     newtio.c_oflag = 0;
77
78     /* set input mode (non-canonical, no echo,...) */
79     newtio.c_lflag = 0;
80
81     newtio.c_cc[VTIME] = 1; /* unsigned inter-unsigned character timer unused
        */
82     newtio.c_cc[VMIN] = 0; /* blocking read until 5 unsigned chars received */
83
84     /*
85      VTIME e VMIN devem ser alterados de forma a proteger com um temporizador
86      a
87      leitura do(s) p[U+FFFD]o(s) caracter(es)
88      */
89     tcflush(fd, TCIOFLUSH);
90
91     if (tcsetattr(fd, TCSANOW, &newtio) == -1) {
92         perror("tcsetattr");
93         exit(-1);
94     }
95
96     //Check conection
97     readControlMessage(fd, C_SET);
98 }
```

```
99     writeControlMessage(fd , C-UA);
100
101     return 0;
102 }
103
104 int llClose(int fd){
105     writeControlMessage(fd , C_DISC);
106
107     readControlMessage(fd , C-UA);
108
109     tcsetattr(fd , TCSANOW, &oldtio);
110
111     close(fd);
112
113     return 0;
114 }
115
116 void readControlMessage(int fd , unsigned char control){
117     unsigned char buf[1];
118     unsigned char message[5];
119     unsigned int res = 0;
120     unsigned int retry = TRUE;
121     unsigned int complete = FALSE;
122     unsigned int state = 0;
123     unsigned int i = 0;
124
125     while (retry == TRUE) {
126         while (complete == FALSE) {
127             res = read(fd , buf , 1);
128             if(res > 0){
129                 switch(state){
130                     case 0:
131                         if(buf[0] == FLAG){
132                             state++;
133                             message[i] = FLAG;
134                             i++;
135                         }
136                         break;
137                     case 1:
138                         if(buf[0] != FLAG){
139                             message[i] = buf[0];
140                             i++;
141                             if(i==4) {
142                                 state++;
143                             }
144                         } else {
145                             i = 0;
146                             state = 0;
147                         }
148                         break;
149                     case 2:
150                         if(buf[0] != FLAG){
151                             i = 0;
```

```
152         state = 0;
153     } else {
154         message[i] = buf[0];
155         complete = TRUE;
156     }
157     break;
158 }
159 }
160 }
161
162 if(message[0] == FLAG
163    && message[1] == A
164    && message[2] == control
165    && message[3] == (A^control)
166    && message[4] == FLAG)
167     retry = FALSE;
168 else if(message[0] == FLAG
169    && message[1] == A
170    && message[2] == C_DISC
171    && message[3] == (A^C_DISC)
172    && message[4] == FLAG){
173     llClose(fd);
174     exit(0);
175 } else {
176     state = 0;
177     complete = FALSE;
178     i = 0;
179 }
180 }
181 }
182
183 void writeControlMessage(int fd, unsigned char control){
184     unsigned char message[5];
185     message[0] = FLAG;
186     message[1] = A;
187     message[2] = control;
188     message[3] = A ^ control;
189     message[4] = FLAG;
190     write(fd, message, 5);
191 }
192
193 int destuffing(unsigned char* buffer, unsigned int packetSize){
194     unsigned char buf, buf2;
195     unsigned char * buffer2 = (unsigned char*)malloc(packetSize * sizeof(
196         unsigned char));
197     if(buffer2 == NULL){
198         printf("Tried to malloc, out of memory\n");
199         exit(-1);
200     }
201     unsigned int newPacketSize = packetSize;
202     unsigned int j = 0;
203     memcpy(buffer2, buffer, packetSize);
```

```
204
205 for(unsigned int i = 0; i < packetSize; i++){
206     buf = *(buffer2 + i);
207     if(buf == ESC){
208         buf2 = *(buffer2 + i + 1);
209         if(buf2 == ESC_FLAG){
210             *(buffer + j) = FLAG;
211         } else if(buf2 == ESC_ESC){
212             *(buffer + j) = ESC;
213         } else {
214             return -1;
215         }
216         newPacketSize--;
217         buffer = (unsigned char *) realloc(buffer, newPacketSize * sizeof(
218             unsigned char));
219         if(buffer == NULL){
220             printf("Tried to realloc, out of memory\n");
221             exit(-1);
222         }
223         i++;
224     } else
225         *(buffer + j) = buf;
226     j++;
227 }
228 changed = buffer;
229
230 free(buffer2);
231 return newPacketSize;
232 }
233
234 int checkBCC2(unsigned char* buffer, unsigned int packetSize){
235     unsigned char bcc2 = *(buffer + packetSize - 1);
236     unsigned char track = *buffer;
237
238     for(unsigned int i = 1; i < packetSize - 1; i++){
239         track ^= *(buffer + i);
240     }
241
242     packetSize--;
243     buffer = (unsigned char*) realloc(buffer, packetSize * sizeof(unsigned char
244         ));
245     if(buffer == NULL){
246         printf("Tried to realloc, out of memory\n");
247         exit(-1);
248     }
249
250     if(track == bcc2)
251         return packetSize;
252
253     changed = buffer;
254
255     return -1;
```

```
255 }
256
257 unsigned int llread(int fd, unsigned char* buffer){
258     unsigned int stop = FALSE;
259     unsigned int state = 0;
260     unsigned char buf, c;
261     unsigned int packetSize = 0;
262     unsigned int res = 0;
263     unsigned int disc = FALSE;
264     packet = -1;
265
266     do {
267
268         message_received = FALSE;
269
270         while(!stop){
271             res = read(fd, &buf, 1);
272             if(res > 0){
273                 switch(state){
274                     case 0: // start
275                         if(buf == FLAG)
276                             state++;
277                         break;
278                     case 1: // address
279                         disc = FALSE;
280                         if(buf == A)
281                             state++;
282                         else
283                             if(buf != FLAG)
284                                 state = 0;
285                         break;
286                     case 2: // control
287                         switch(buf){
288                             case C_0:
289                                 packet = 0;
290                                 c = buf;
291                                 state++;
292                                 break;
293                             case C_1:
294                                 packet = 1;
295                                 c = buf;
296                                 state++;
297                                 break;
298                             case C_DISC:
299                                 c = buf;
300                                 state++;
301                                 disc = TRUE;
302                                 break;
303                             case FLAG:
304                                 state = 1;
305                                 break;
306                             default:
307                                 state = 0;
```

```
308         break;
309     }
310     break;
311     case 3: // bcc1
312         if (buf == (A ^ c)){
313             if(disc){
314                 state = 5;
315                 disc = FALSE;
316             } else
317                 state++;
318         } else
319             if(buf == FLAG)
320                 state = 1;
321             else
322                 state = 0;
323         break;
324     case 4: //data
325         if (buf == FLAG) {
326             stop = TRUE;
327         } else {
328             packetSize++;
329             buffer = (unsigned char *) realloc(buffer, packetSize * sizeof(
unsigned char));
330             if(buffer == NULL){
331                 printf("Tried to realloc, out of memory\n");
332                 exit(-1);
333             }
334             *(buffer + packetSize - 1) = buf;
335         }
336         break;
337     case 5: //disc
338         if (buf == FLAG) {
339             llClose(fd);
340             exit(0);
341         } else
342             state = 0;
343         break;
344     }}
345 }
346
347 numPackets++;
348 packetSize = destuffing(buffer, packetSize);
349 buffer = changed;
350 packetSize = checkBCC2(buffer, packetSize);
351 buffer = changed;
352 if(packetSize != -1){
353     if(packet == 0)
354         writeControlMessage(fd, RR_C_1);
355     else
356         writeControlMessage(fd, RR_C_0);
357
358     message_received = TRUE;
359     alarm(0);
```

```
360
361     if (packet == expected) {
362         numRR++;
363         expected ^= 1;
364     }
365     else
366         packetSize = -1;
367 } else {
368     if(packet == 0) {
369         writeControlMessage(fd, REJ_C_1);
370     } else {
371         writeControlMessage(fd, REJ_C_0);
372     }
373
374     numREJ++;
375
376 }
377 } while(packetSize == -1);
378
379 return packetSize;
380 }
381
382 int getFileInfo(unsigned char* start){
383     unsigned char type = *(start);
384
385     if(type != START)
386         return -1;
387
388     unsigned char param = *(start + 1);
389     unsigned int octets = (unsigned int)*(start + 2);
390     off_t octetVal;
391     off_t size = 0;
392
393     if(param != SIZE)
394         return -1;
395
396     for(unsigned int i = 0; i < octets; i++) {
397         octetVal = (*(start + 3 + i) << ((octets-i-1) * 8));
398         size = size | octetVal;
399     }
400
401     info.size = (unsigned int)size;
402
403     unsigned char* next = start + 3 + octets;
404     param = *(next);
405
406     if(param != NAME)
407         return -1;
408
409     octets = (unsigned int)*(next + 1);
410
411     unsigned char* name = (unsigned char*)malloc((octets+1) * sizeof(unsigned
char));
```

```
412     if(name == NULL){
413         printf("Tried to malloc, out of memory\n");
414         exit(-1);
415     }
416
417     int i;
418     for(i = 0; i < octets; i++) {
419         *(name + i) = *(next + 2 + i);
420     }
421
422     *(name + i) = '\0';
423
424     info.name = name;
425
426     return 0;
427 }
428
429 unsigned int isEndPacket(unsigned char* start, unsigned int startSize,
430                          unsigned char* end, unsigned int endSize) {
431
432     unsigned char type = *(end);
433
434     if(startSize != endSize || type != END)
435         return FALSE;
436
437     for(unsigned int i = 1; i < startSize; i++) {
438         if (*(start + i) != *(end + i))
439             return FALSE;
440     }
441
442     return TRUE;
443 }
444
445 unsigned int removeHeader(unsigned char* packet, unsigned int size){
446     unsigned int newSize = size - 4;
447     unsigned char *newPacket = (unsigned char*)malloc(newSize * sizeof(unsigned
448         char));
449
450     if(newPacket == NULL){
451         printf("Tried to malloc, out of memory\n");
452         exit(-1);
453     }
454
455     for (unsigned int i = 0; i < newSize; i++)
456     {
457         *(newPacket + i) = *(packet + i + 4);
458     }
459
460     changed = newPacket;
461
462     free(packet);
```



```
463     return newSize;
464 }
465
466 void readContent(int fd, unsigned char* start, unsigned int startSize){
467     unsigned char* packet;
468     unsigned int packetSize;
469     unsigned int index = 0;
470
471     while(TRUE) {
472
473         packet = (unsigned char*)malloc(sizeof(unsigned char));
474         if(packet == NULL){
475             printf("Tried to malloc, out of memory\n");
476             exit(-1);
477         }
478
479         packetSize = llread(fd, packet);
480         packet = changed;
481
482         if(isEndPacket(start, startSize, packet, packetSize)){
483             break;
484         }
485
486         packetSize = removeHeader(packet, packetSize);
487         packet = changed;
488
489
490         memcpy(info.content + index, packet, packetSize);
491         index += packetSize;
492
493         free(packet);
494
495     }
496
497     if (alarm_flag) {
498         printf("[readContent] Timed out, exiting application\n");
499         exit(1);
500     }
501 }
502
503 void createFile(){
504     FILE* file = fopen((unsigned char*)info.name, "wb+");
505     fwrite((unsigned char*)info.content, sizeof(unsigned char), info.size, file);
506     fclose(file);
507 }
508 }
```

**transmitter.h**

```
1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <fcntl.h>
4 #include <termios.h>
5 #include <stdio.h>
6 #include <signal.h>
7 #include <stdlib.h>
8 #include <unistd.h>
9 #include <string.h>
10 #include <time.h>
11
12 #define TIMEOUT          3
13
14 #define BAUDRATEDEFAULT  B38400
15 #define MODEMDEVICE       "/dev/ttyS1"
16 #define _POSIX_SOURCE      1 /* POSIX compliant source */
17 #define FALSE              0
18 #define TRUE               1
19
20 #define MAXALARMCOUNT 3
21
22 #define FLAG               0x7e
23 #define ADDRESS            0x03
24 #define CONTROLSET         0x03
25 #define CONTROLUA          0x07
26 #define CONTROL_DISC       0x0b
27 #define BCC_UA              (ADDRESS ^ CONTROLUA)
28 #define PACKET_HEADER_SIZE 6 //Number of bytes used around an information
    packet
29
30 #define ESCAPE_CODE         0x7d
31 #define STUFF_FLAG_CODE     0x5e
32 #define STUFF_ESCAPE_CODE   0x5d
33
34 #define A_WRITER            0x03
35
36 #define RR_0                0x05
37 #define RR_1                0x85
38 #define REJ_0               0x01
39 #define REJ_1               0x81
40
41 #define APP_CONTROL_SIZE_CONST 0x09
42 #define APP_CONTROL_DATA    0x01
43 #define APP_CONTROL_START   0x02
44 #define APP_CONTROL_END     0x03
45 #define APP_T_FILESIZE      0x00
46 #define APP_T_FILENAME      0x01
47 #define APP_L_FILESIZE      0x04
48
49 #define C_0                  0x00
```

```
50 #define C_1          0x40
51
52 #define PACKET_SIZE_DEFAULT 255
53
54 void set_alarm();
55 void remove_alarm();
56 unsigned char * openFile(unsigned char * file, int * fileSize);
57 int sendFile(int fd, unsigned char * fileName, int fileNameSize);
58 unsigned char * prepareAppControlPacket(unsigned char control, int fileSize,
    unsigned char * fileName, int fileNameSize, int * appControlPacketSize);
59 unsigned char * prepareDataPacketHeader(unsigned char * data, int fileSize,
    int * packetSize, int * numPackets);
60 unsigned char * splitData(unsigned char * fileData, int fileSize, int *
    currPos, int * currPacketSize);
61 int llopen(int fd);
62 int llclose(int fd);
63 int sendControlMessage(int fd, unsigned char control);
64 int stopAndWaitControl(int fd, unsigned char control_sent, unsigned char
    control_expected);
65 int stopAndWaitData(int fd, unsigned char * buffer, int length);
66 void stateMachine(unsigned char *message, int *state, unsigned char control);
67 int llopen(int fd);
68 int llwrite(int fd, unsigned char * buffer, int length);
69 unsigned char * concat(const unsigned char * s1, const unsigned char * s2);
70 unsigned char * packetStuffing(unsigned char * message, int size, int *
    finalPacketSize);
71 unsigned char * BCC2Stuffing(unsigned char * bcc2, int * stuffedBCC2Size);
72 unsigned char * generateBCC2(unsigned char *message, int sizeOfMessage);
73 unsigned char * preparePacket(unsigned char * buf, unsigned char * bcc2, int
    * finalPacketSize, int * stuffedBCC2Size);
```

**transmitter.c**

```
1  /*Non-Canonical Input Processing*/
2  #include "transmitter.h"
3
4  int packetSign = C_0;
5  volatile int STOP = FALSE;
6  int alarm_flag = FALSE;
7  int alarm_counter = 0;
8  int message_received = FALSE;
9  int numPackets = 0;
10 int numRR = 0;
11 int numREJ = 0;
12 int activeBaudrate = BAUDRATE_DEFAULT;
13 int activePacketSize = PACKET_SIZE_DEFAULT;
14 int testMode = FALSE;
15
16 //
17 // Alarm handler
18 //
19
20 void alarm_handler() {
21     alarm_flag = TRUE;
22     alarm_counter++;
23 }
24
25 void reset_alarm_flag(){
26     alarm_flag = FALSE;
27 }
28
29 void reset_alarm_counter() {
30     alarm_counter = 0;
31 }
32
33 //
34 // Application
35 //
36
37
38 int main(int argc, char** argv)
39 {
40     int fd;
41     struct termios oldtio, newtio;
42
43     if ( (argc < 3) ||
44         ((strcmp("/dev/ttyS0", argv[1])!=0) &&
45          (strcmp("/dev/ttyS1", argv[1])!=0) )) {
46         printf("Usage:\nserial SerialPort\n\tex: nserial /dev/ttyS1 test.txt\n");
47         exit(1);
48     }
49 }
```

```
50     if (argc == 5) {
51         activeBaudrate = (int) strtol(argv[3], NULL, 10);
52         activePacketSize = (int) strtol(argv[4], NULL, 10);
53         testMode = TRUE;
54
55         printf("Test mode active\n**Current settings**\nBaudrate: %i\nPacket
56         size: %i\n", activeBaudrate, activePacketSize);
57     }
58
59     /*
60     Open serial port device for reading and writing and not as controlling
61     tty
62     because we don't want to get killed if linenoise sends CTRL-C.
63     */
64
65     fd = open(argv[1], ORDWR | O_NOCTTY);
66     if (fd < 0) {perror(argv[1]); exit(-1); }
67
68     //printf("Serial port open\n");
69
70     if ( tcgetattr(fd,&oldtio) == -1) { /* save current port settings */
71         perror("tcgetattr");
72         exit(-1);
73     }
74
75     bzero(&newtio, sizeof(newtio));
76     newtio.c_cflag = activeBaudrate | CS8 | CLOCAL | CREAD;
77     newtio.c_iflag = IGNPAR;
78     newtio.c_oflag = 0;
79
80     /* set input mode (non-canonical, no echo,...) */
81
82     newtio.c_lflag = 0;
83
84     newtio.c_cc[VTIME] = 1; /* inter-character timer unused */
85     newtio.c_cc[VMIN] = 5; /* blocking read until 5 chars received */
86
87     /*
88     VTIME e VMIN devem ser alterados de forma a proteger com um temporizador
89     a
90     leitura do(s) proximo(s) caracter(es)
91     */
92
93     tcflush(fd, TCIOFLUSH);
94
95     if ( tcsetattr(fd,TCSANOW,&newtio) == -1) {
96         perror("tcsetattr");
97         exit(-1);
98     }
99
100    //printf("New termios structure set\n");
```

```
100
101 //Set the alarm
102
103 struct sigaction sa;
104 sa.sa_handler = alarm_handler;
105 sigemptyset(&sa.sa_mask);
106 sa.sa_flags = 0;
107
108 sigaction(SIGALRM, &sa, NULL);
109
110 //
111 //Timer for efficiency test
112 //
113
114 struct timespec transmitionStart, transmitionEnd;
115 clock_gettime(CLOCK_REALTIME, &transmitionStart);
116
117 //
118 //llopen
119 //
120
121 int error = llopen(fd);
122 if(error != 0){
123     perror("Could not connect\n");
124     exit(1);
125 }
126
127 //
128 //send file
129 //
130
131 //printf("[main] File name size: %i\n", (int) strlen(argv[2]));
132 sendFile(fd, (unsigned char *) argv[2], strlen(argv[2]));
133
134 //
135 //llclose
136 //
137
138 llclose(fd);
139
140 clock_gettime(CLOCK_REALTIME, &transmitionEnd);
141
142 double deltaTime = (transmitionEnd.tv_sec - transmitionStart.tv_sec) + (
    transmitionEnd.tv_nsec - transmitionStart.tv_nsec) / 1E9;
143
144 if (testMode) {
145     printf("Number of packets sent: %i\nNumber of packets rejected: %i\n", numPackets, numREJ);
146     printf("Transmission time: %f\n", deltaTime);
147 }
148
149 if (tcsetattr(fd, TCSANOW, &oldtio) == -1) {
150     perror("tcsetattr");
151 }
```

```
151     exit(-1);
152 }
153
154 close(fd);
155 return 0;
156 }
157
158 //
159 // Application functions
160 //
161
162 int sendFile(int fd, unsigned char * fileName, int fileNameSize) {
163
164     //open file
165
166     int fileSize = 0;
167     int appControlPacketSize = 0;
168     unsigned char * fileData = openFile(fileName, &fileSize);
169
170     if (testMode) {
171         printf("File size: %i bytes (%i bits)\n", fileSize, fileSize*8);
172     }
173
174     //prepare and send start packet
175
176     unsigned char * startPacket = prepareAppControlPacket(APP_CONTROL_START,
177         fileSize, fileName, fileNameSize, &appControlPacketSize);
178
179     stopAndWaitData(fd, startPacket, appControlPacketSize);
180
181     //free allocated memory (startPacket)
182
183     free(startPacket);
184
185     //send the file
186
187     int currPos = 0;
188     int currPacketSize = activePacketSize;
189     int numPackets = 0;
190     unsigned char * currPacket;
191
192     while(currPos < fileSize) {
193         currPacket = splitData(fileData, fileSize, &currPos, &currPacketSize);
194         currPacket = prepareDataPacketHeader(currPacket, fileSize, &
195             currPacketSize, &numPackets);
196
197         int stopAndWaitDataReturn;
198
199         stopAndWaitDataReturn = stopAndWaitData(fd, currPacket, currPacketSize);
200         switch(stopAndWaitDataReturn) {
201             case 0:
202                 break;
203             case 1:
```

```
202     //timed out, exit application
203     perror("[sendFile] Connection timed out, closing application\n");
204     exit(1);
205     case 2:
206         //packet rejected, send it again
207         stopAndWaitData(fd, currPacket, currPacketSize);
208     }
209
210     currPacketSize = activePacketSize;
211     free(currPacket);
212 }
213
214 //send end packet
215
216 unsigned char * endPacket = prepareAppControlPacket(APP_CONTROLEND,
    fileSize, fileName, fileNameSize, &appControlPacketSize);
217
218 stopAndWaitData(fd, endPacket, appControlPacketSize);
219
220 printf("[sendFile] File sent\n");
221
222 //free allocated memory (endPacket and fileData)
223 free(fileData);
224 free(startPacket);
225
226 return 0;
227 }
228
229 unsigned char * openFile(unsigned char * file, int * fileSize) {
230
231     FILE * f;
232     struct stat metadata;
233     unsigned char * fileData;
234
235     if ((f = fopen((char *) file, "rb")) == NULL) {
236         perror("[openFile] Error opening file\n");
237         exit(1);
238     }
239
240     stat((char *) file, &metadata);
241     (*fileSize) = metadata.st_size;
242
243     fileData = (unsigned char *) malloc((*fileSize)*sizeof(unsigned char));
244     fread(fileData, sizeof(unsigned char), *fileSize, f);
245     return fileData;
246 }
247
248 unsigned char * prepareAppControlPacket(unsigned char control, int fileSize,
    unsigned char * fileName, int fileNameSize, int * appControlPacketSize) {
249
250     (*appControlPacketSize) = APP_CONTROL_SIZE_CONST + fileNameSize;
251     unsigned char * controlPacket = (unsigned char *) malloc ((*
    appControlPacketSize)*sizeof(unsigned char));
```



```
252
253     controlPacket[0] = control;
254     controlPacket[1] = APP_T_FILESIZE;
255     controlPacket[2] = APP_L_FILESIZE;
256     controlPacket[3] = (fileSize >> 24) & 0xFF;
257     controlPacket[4] = (fileSize >> 16) & 0xFF;
258     controlPacket[5] = (fileSize >> 8) & 0xFF;
259     controlPacket[6] = fileSize & 0xFF;
260     controlPacket[7] = APP_T_FILENAME;
261     controlPacket[8] = fileNameSize;
262
263     int i;
264     for (i = 0; i < fileNameSize; i++) {
265         controlPacket[9+i] = fileName[i];
266     }
267
268     return controlPacket;
269 }
270
271
272 unsigned char * prepareDataPacketHeader(unsigned char * data, int fileSize,
273                                         int * packetSize, int * numPackets) {
274
275     unsigned char * finalDataPacket = (unsigned char *) malloc((fileSize+4) *
276                                                                sizeof(unsigned char));
277
278     finalDataPacket[0] = APP_CONTROLDATA;
279     finalDataPacket[1] = (*numPackets) % 255;
280     finalDataPacket[2] = fileSize / 256;
281     finalDataPacket[3] = fileSize % 256;
282
283     memcpy(finalDataPacket+4, data, (*packetSize)*sizeof(unsigned char));
284     (*packetSize) += 4;
285
286     (*numPackets)++;
287     return finalDataPacket;
288 }
289
290 unsigned char * splitData(unsigned char * fileData, int fileSize, int *
291                           currPos, int * currPacketSize) {
292
293     unsigned char * packet;
294     int j = (*currPos);
295
296     if (j + (*currPacketSize) > fileSize) {
297         (*currPacketSize) = (fileSize - j);
298     }
299
300     packet = (unsigned char *) malloc((*currPacketSize)*sizeof(unsigned char));
301
302     int i;
303     for(i = 0; i < (*currPacketSize); i++, j++) {
```

```

302     packet[i] = fileData[j];
303 }
304
305 (* currPos) = j;
306
307 return packet;
308
309 }
310
311
312 //
313 // Data link layer functions
314 //
315
316 int sendControlMessage(int fd, unsigned char control) {
317
318     int res;
319     unsigned char message[5];
320
321     message[0] = FLAG;
322     message[1] = ADDRESS;
323     message[2] = control;
324     message[3] = message[1] ^ message[2];
325     message[4] = FLAG;
326
327     res = write(fd, message, 5);
328     if(res <= 0){
329         return FALSE;
330     }
331     else{
332         //printf("[sendControlMessage] Message sent: 0x%02x\n", message[2]);
333         return TRUE;
334     }
335 }
336
337 /**
338  * Recebe uma mensagem e um estado e interpreta em que parte da leitura da
339  * trama de supervisao de [U+FFFD]
340  * @param message - Mensagem para ser interpretada
341  * @param state - Estado/ps[U+FFFD] na interpreta[U+FFFD]ao da trama
342  */
343 void stateMachine(unsigned char *message, int *state, unsigned char control){
344     switch(*state){
345
346         case 0: /* Esta a espera do inicio da trama
347         (0x7E) */
348             if(*message == FLAG){
349                 *state = 1;
350             }
351             break;
352
353         case 1:

```

```

353         if(*message == ADDRESS){                /* Houve um erro e Address incorreto */
354             *state = 2;
355         }
356         else if(*message == FLAG){                /* Se tiver um 0x7E no meio da trama */
357             *state = 1;
358         }
359         else{                                     /* Houve um erro e Address incorreto */
360             *state = 0;
361         }
362         break;
363
364     case 2:
365         if(*message == control){                  /* Recebe o valor de Controlo */
366             *state = 3;
367         }
368         else if(*message == FLAG){                /* Se tiver um 0x7E no meio da trama */
369             *state = 1;
370         }
371         else{                                     /* Houve um erro e Controlo incorreto */
372             *state = 0;
373         }
374         break;
375
376     case 3:
377         if(*message == (control ^ ADDRESS)){      /* BCC lido com sucesso */
378             *state = 4;
379         }
380         else{                                     /* Erro com BCC */
381             *state = 0;
382         }
383         break;
384
385     case 4:
386         if(*message == FLAG){                     /* Recebe o ultimo 7E */
387             message_received = TRUE;
388             alarm(0);
389             //printf("[stateMachine] Received 0x%02x\n", control);
390         }
391         else{                                     /* Erro no 7E */
392             *state = 0;
393         }
394         break;
395     }
396 }
397
398 int stopAndWaitControl(int fd, unsigned char control_sent, unsigned char
399                        control_expected) {
400
401     int res = 0;
402     unsigned char buf[1];

```

```

402     int state = 0;
403
404     reset_alarm_flag();
405     reset_alarm_counter();
406     message_received = FALSE;
407
408     while(alarm_counter < MAX_ALARM_COUNT && !message_received){
409
410         sendControlMessage(fd, control_sent);
411         alarm(TIMEOUT);
412         state = 0;
413         while(!alarm_flag && !message_received) {
414             res = read(fd, buf, 1);
415             if(res <= 0){
416                 //perror("stopAndWait: read nothing\n");
417             }
418             else
419             {
420                 //printf("[stopAndWaitControl] Received answer: 0x%02x\n", *buf);
421             }
422             stateMachine(buf, &state, control_expected);
423         }
424
425         reset_alarm_flag();
426     };
427
428     if(alarm_counter < MAX_ALARM_COUNT){
429         return 0;
430     }
431     else
432         return 1;
433 }
434
435 void stateMachineData(unsigned char *message, int *state, unsigned char *
    controlReceived){
436     switch(*state){
437
438         case 0:                                     /* Esta a espera do inicio da trama
    (0x7E) */
439             if(*message == FLAG){
440                 *state = 1;
441             }
442             break;
443
444         case 1:
445             if(*message == ADDRESS){                 /* Houve um erro e Address e [U+FFFD]
    espera do A */
446                 *state = 2;
447             }
448             else if(*message == FLAG){               /* Se tiver um 0x7E no meio da trama */
449                 *state = 1;
450             }
451             else{                                     /* Houve um erro e Address e [U+FFFD]
    incorreto */

```

```

452         *state = 0;
453     }
454     break;
455
456     case 2:
457         if(*message == RR_0 || *message == RR_1 || *message == REJ_0 || *
message == REJ_1){          /* Recebe o valor de Controlo */
458             *controlReceived = (*message);
459             *state = 3;
460         }
461         else if(*message == FLAG){    /* Se tiver um 0x7E no meio da trama */
462             *state = 1;
463         }
464         else{                          /* Houve um erro e Controlo[U+FFFD]
[U+FFFD]correto */
465             *state = 0;
466         }
467         break;
468
469     case 3:
470         if(*message == ((*controlReceived) ^ ADDRESS)){          /* BCC
lido com sucesso */
471             *state = 4;
472         }
473         else{                          /* Erro com BCC */
474             *state = 0;
475         }
476         break;
477
478     case 4:
479         if(*message == FLAG){          /* Recebe o ultimo 7E */
480             message_received = TRUE;
481             alarm(0);
482         }
483         else{                          /* Erro no 7E */
484             *state = 0;
485         }
486         break;
487     }
488 }
489
490 int stopAndWaitData(int fd, unsigned char * buffer, int length) {
491
492     int res = 0;
493     unsigned char buf[1];
494     int state = 0;
495     unsigned char controlReceived[1];
496
497     reset_alarm_flag();
498     reset_alarm_counter();
499     message_received = FALSE;
500
501     //send the message

```

```
502 llwrite(fd, buffer, length);
503
504
505
506 while(alarm_counter < MAXALARMCOUNT && !message_received){
507
508     alarm(TIMEOUT);
509     while(!alarm_flag && !message_received) {
510         res = read(fd, buf, 1);
511         if(res <= 0){
512             //perror("stopAndWait: read nothing\n");
513         }
514         else
515         {
516             //printf("[stopAndWaitData] Received answer: 0x%02x\n", *buf);
517             stateMachineData(buf, &state, controlReceived);
518         }
519     }
520
521     //re-send the data packet
522     if (alarm_flag) {
523         llwrite(fd, buffer, length);
524     }
525
526     reset_alarm_flag();
527
528 };
529
530 if(alarm_counter < MAXALARMCOUNT) {
531     switch(controlReceived[0]) {
532     case RR_0:
533         packetSign = C_0;
534         numRR++;
535         //printf("rr0 new sign is %c\n", packetSign);
536         return 0;
537     case RR_1:
538         packetSign = C_1;
539         numRR++;
540         //printf("rr1 new sign is %c\n", packetSign);
541         return 0;
542     case REJ_0:
543         numREJ++;
544         //printf("rej0\n");
545         return 2;
546     case REJ_1:
547         numREJ++;
548         //printf("rej1\n");
549         return 2;
550     }
551 } else {
552     printf("[stopAndWaitData] Timeout while sending data, terminating program\n");
553     exit(1);
```

```
554     }
555
556     return 1;
557
558 }
559
560 int llopen(int fd){
561     return stopAndWaitControl(fd , CONTROLSET,CONTROLUA);
562 }
563
564 int llclose(int fd) {
565     if (stopAndWaitControl(fd , CONTROLDISC,CONTROLDISC)) {
566         return 1;
567     }
568
569     if (sendControlMessage(fd , CONTROLUA)) {
570         return 1;
571     }
572     return 0;
573 }
574
575 /**
576 * Calculate the parity of the message of the argument
577 * @param message – pointer to the message which parity wants to be calculated
578 * @param size – size of the message sent
579 * @returns the parity of the message in form of a unsigned char
580 */
581 unsigned char generateBCC2(unsigned char *message , int sizeOfMessage){
582
583     unsigned char result = message[0];
584
585     int i;
586     for(i = 1; i < sizeOfMessage; i++){
587         result = result ^ message[i];
588     }
589
590     return result;
591 }
592
593 /**
594 *
595 */
596 int llwrite(int fd , unsigned char * buffer , int length) {
597
598     int finalPacketSize = 0;
599     int stuffedBCC2Size = 0;
600     unsigned char * finalPacket;
601
602     //
603     //bcc2
604     //
605
606     unsigned char BCC2 = generateBCC2(buffer , length);
```

```
607 unsigned char * stuffedBCC2 = (unsigned char *) malloc(sizeof(unsigned char
    ));
608
609 //
610 //stuffing
611 //
612
613 stuffedBCC2 = BCC2Stuffing(&BCC2, &stuffedBCC2Size);
614 finalPacket = packetStuffing(buffer, length, &finalPacketSize);
615
616
617 //
618 //concatenate packet header, stuffed message and packet trailer
619 //
620
621 finalPacket = preparePacket(finalPacket, stuffedBCC2, &finalPacketSize, &
    stuffedBCC2Size);
622
623 //
624 //sending packet
625 //
626
627 write(fd, finalPacket, finalPacketSize);
628
629 numPackets++;
630
631 free(stuffedBCC2);
632 free(finalPacket);
633
634 return 0;
635 }
636
637 unsigned char * BCC2Stuffing(unsigned char * bcc2, int * stuffedBCC2Size) {
638
639 unsigned char * stuffedBCC2 = (unsigned char *) malloc(sizeof(unsigned char
    ));
640
641 if((*bcc2) == FLAG){
642     stuffedBCC2 = (unsigned char *)realloc(stuffedBCC2, 2*sizeof(unsigned char
    ));
643     stuffedBCC2[0] = ESCAPE_CODE;
644     stuffedBCC2[1] = STUFF_FLAG.CODE;
645     (*stuffedBCC2Size) = 2;
646 }
647 else if((*bcc2) == ESCAPE_CODE) {
648     stuffedBCC2 = (unsigned char *)realloc(stuffedBCC2, 2*sizeof(unsigned char
    ));
649     stuffedBCC2[0] = ESCAPE_CODE;
650     stuffedBCC2[1] = STUFF_ESCAPE_CODE;
651     (*stuffedBCC2Size) = 2;
652 }
653 else {
654     stuffedBCC2[0] = (*bcc2);
```



```
655     (*stuffedBCC2Size) = 1;
656 }
657
658 return stuffedBCC2;
659 }
660
661 /**
662 *   Stuffs the message in buf to prevent it from having the char FLAG which
663 *   would cause a problem while reading
664 *   @param buf - message to be stuffed
665 *   @param len - length of the message to be stuffed
666 *   @return the message but stuffed (with no FLAGS in it)
667 */
668 unsigned char * packetStuffing(unsigned char * message, int size, int *
669                               finalPacketSize) {
670
671     //Counter for the original message
672     int i;
673     //Counter for the new message
674     int j = 0;
675
676     int sizeFinalMessage = size;
677     unsigned char *finalMessage = (unsigned char *)malloc(sizeof(unsigned char)
678 *size);
679
680     if (finalMessage == NULL) {
681         printf("[packetStuffing] Couldn't allocate memory\n");
682     }
683
684     //Loop through all the old message and generate the new one
685     for(i = 0; i < size ; i++) {
686
687         if(message[i] == FLAG) {
688             sizeFinalMessage++;
689             finalMessage = (unsigned char *)realloc(finalMessage, sizeFinalMessage *
690 sizeof(unsigned char));
691             finalMessage[j] = ESCAPE_CODE;
692             finalMessage[j+1] = STUFF_FLAG_CODE;
693             j += 2;
694         }
695         else if(message[i] == ESCAPE_CODE) {
696             sizeFinalMessage++;
697             finalMessage = (unsigned char *)realloc(finalMessage, sizeFinalMessage *
698 sizeof(unsigned char));
699             finalMessage[j] = ESCAPE_CODE;
700             finalMessage[j+1] = STUFF_ESCAPE_CODE;
701             j += 2;
702         }
703         else {
704             finalMessage[j] = message[i];
705             j++;
706         }
707     }
708 }
```

```
703
704     *finalPacketSize = sizeFinalMessage;
705     return finalMessage;
706 }
707
708 unsigned char * preparePacket(unsigned char * buf, unsigned char * bcc2, int
    * finalPacketSize, int * stuffedBCC2Size) {
709
710     unsigned char header[4];
711     header[0] = FLAG;
712     header[1] = A_WRITER;
713     header[2] = packetSign;
714     header[3] = header[1] ^ header[2];
715
716     unsigned char trailer[1];
717     trailer[0] = FLAG;
718
719     int auxFinalPacketSize = ((*finalPacketSize) + 5 + (*stuffedBCC2Size));
720
721     unsigned char * auxBuf = (unsigned char *) malloc(auxFinalPacketSize *
        sizeof(unsigned char));
722
723     memcpy(auxBuf, header, 4*sizeof(unsigned char));
724     memcpy(auxBuf+4, buf, (*finalPacketSize)*sizeof(unsigned char));
725     memcpy(auxBuf+4+(*finalPacketSize), bcc2, (*stuffedBCC2Size)*sizeof(
        unsigned char));
726     memcpy(auxBuf+4+(*finalPacketSize)+(*stuffedBCC2Size), trailer, 1*sizeof(
        unsigned char));
727
728     (*finalPacketSize) = auxFinalPacketSize; // flag, address, control, bcc1,
        bcc2, flag
729
730     free(buf);
731     return auxBuf;
732
733 }
```