

# Primeiro Trabalho Laboratorial

Hugo Guimarães, Pedro Ponte

November 16, 2020

## Sumário

Este trabalho foi realizado no contexto da cadeira Redes de Computadores, com o objetivo de implementar um protocolo de ligação de dados através de uma porta série, permitindo a transmissão de um ficheiro entre 2 computadores.

Deste modo, o trabalho foi concluído com sucesso, dado que foi possível implementar uma aplicação que cumprisse os objetivos estabelecidos.

## 1 Introdução

Este trabalho pretende implementar um protocolo de ligação de dados baseado no guião fornecido, de modo a ser possível transferir ficheiros através de uma porta série.

O relatório pretende descrever detalhadamente a aplicação implementada, estando dividida nas seguintes secções:

**Arquitetura** Descrição dos blocos funcionais e interfaces implementados.

**Estrutura do Código** Descrição das APIs, principais estruturas de dados, principais funções e a sua relação com a arquitetura.

**Casos De Uso Principais** Identificação dos principais casos de uso e da sequência de chamada de funções.

**Protocolo De Ligação Lógica** Descrição dos principais aspetos funcionais do protocolo de ligação lógica e da sua estratégia de implementação com apresentação de extratos de código.

**Protocolo De Aplicação** Descrição dos principais aspetos funcionais do protocolo de aplicação e da sua estratégia de implementação com apresentação de extratos de código.

**Validação** Descrição dos testes efetuados com apresentação quantificada dos resultados.

**Eficiência Do Protocolo De Ligação De Dados** Caracterização estatística da eficiência do protocolo de Stop&Wait implementado.

**Conclusões** Síntese da informação apresentada nas secções anteriores e reflexão sobre os objetivos de aprendizagem alcançados.

## 2 Arquitetura

O trabalho está dividido em 2 secções fundamentais, o emissor e o recetor. Ambos incorporam a sua própria camada de ligação de dados e aplicação.

## 3 Estrutura do Código

O código está dividido em vários ficheiros.

Os ficheiros *llfunctions.c* e *stateMachines.c* são responsáveis pelo tratamento do protocolo da ligação de dados, sendo o *stateMachines.c* unicamente responsável pela implementação das máquinas de estado de aceitação de mensagens.

O ficheiro *application.c* é responsável pelo tratamento do protocolo de aplicação.

Os ficheiros *emissor.c* e *recetor.c* são responsáveis pelo fluxo de execução do programa, dos lados do emissor e recetor, respetivamente. Ambos contêm apenas a função *main* e todas as funções chamadas estão implementadas nos restantes ficheiros.

### **emissor.c**

- **main** - Controla os processos ao nível da camada da aplicação da parte do emissor e faz as chamadas às funções da camada de ligação.

### **recetor.c**

- **main** - Controla os processos ao nível da camada da aplicação da parte do recetor e faz as chamadas às funções da camada de ligação.

### **llfunctions.c**

- **llopen** - Do lado do emissor, envia uma trama de supervisão *SET* e recebe uma trama *UA*, enquanto no lado do recetor este espera pela trama de controlo *SET* enviada pelo emissor e responde com uma trama *UA*.
- **llclose** - Do lado do emissor, envia uma trama de supervisão *DISC*, espera que o emissor responda com uma trama *DISC* e envia uma trama *UA*. No lado do recetor, este aguarda pela trama *DISC* enviada pelo emissor, responde com uma trama *DISC* e depois recebe uma trama *UA*.
- **llwrite** - Faz o stuffing das tramas I e envia-as, recebendo *REJ* ou *RR* como resposta.

- **llread** - Lê as tramas I enviadas pelo llwrite e envia uma resposta do tipo *RR*, no caso das tramas I recebidas sem erros detetados no cabeçalho e no campo de dados, ou do tipo *REJ*, no caso das tramas I sem erro detetado no cabeçalho, mas com erros no campo de dados.
- **alarmHandler** - Substituição do handler do alarme para permitir que as tramas sejam enviadas *MAXTRIES* vezes em situação de erro.
- **getBcc2** - Gera o *BCC2* no lado do emissor.
- **stuffBCC2** - Realiza o stuff do *BCC2* no lado do emissor, após a geração do *BCC*.

#### **stateMachines.c**

- **sendMessage** - Faz o parse da trama e envia-a pela porta de série.
- **readSetMessage** - Máquina de estados que recebe a trama *SET* e verifica a sua correção.
- **readReceiverMessage** - Recebe as tramas *REJ* e *RR* enviadas pelo recetor e verifica a sua correção.
- **receiveUA** - Recebe as tramas *UA* e verifica a sua correção.
- **receiverRead\_StateMachine** - Recebe as tramas I enviadas pelo emissor, verifica a sua correção, efetua o destuffing necessário, guarda os dados contidos nas tramas I num novo array e envia uma trama *REJ* ou *RR* como resposta, dependendo da ocorrência de erros nas tramas recebidas ou no respetivo destuffing.
- **receiveDISC** - Recebe as tramas *DISC* e verifica a sua correção.
- **checkBCC2** - Verifica a correção do *BCC2* no lado do recetor.

#### **application.c**

- **openFile** - Abre o ficheiro enviado como argumento e obtém o seu conteúdo e tamanho.
- **parseControlPacket** - Gera o pacote de controlo de um ficheiro para depois ser enviado.
- **parseDataPacket** - Codifica a mensagem num pacote de acordo com o protocolo estabelecido.

- **splitPacket** - Obtém uma porção da mensagem, de modo a enviar os dados sob a forma de uma trama I.
- **checkStart\_StateMachine** - Verifica se o primeiro pacote recebido pelo recetor é de facto o pacote de controlo de início.
- **checkEND** - Verifica se o pacote de controlo inicial é igual ao final.
- **assembleDataPacket** - Obtém os dados enviados pelo emissor através do pacote recebido pela porta série.
- **createFile** - Cria o ficheiro final após ter lido toda a informação através da porta série.

## 4 Casos De Uso Principais

Este trabalho laboratorial tem 2 casos de uso distintos: a interface e a transmissão do ficheiro. A interface permite ao utilizador iniciar a aplicação. No lado do emissor, selecciona a porta de série que pretende utilizar (ex: `/dev/ttyS0`) e o ficheiro que pretende enviar (ex: **pinguim.png**). Do lado do recetor, basta apenas seleccionar a porta de série a ser utilizada.

A transmissão do ficheiro, através da porta de série, entre os dois dispositivos, permite o estabelecimento da ligação entre os dispositivos, sendo o emissor responsável pela escolha do ficheiro a enviar. o emissor envia, trama a trama, os dados, sendo recebidos de igual forma pelo recetor que, antes de terminar a ligação, origina um ficheiro igual ao recebido originalmente.

- Configuração da ligação e escolha do ficheiro a ser enviado pelo emissor;
- Estabelecimento da ligação entre o emissor e o recetor;
- Envio, trama a trama, dos dados por parte do emissor;
- Receção dos dados enviados pelo recetor, que os guarda num ficheiro com o mesmo nome do original à medida que os vai recebendo;
- Terminação da ligação.

## 5 Protocolo De Ligação Lógica

O objetivo do protocolo de ligação lógica é estabelecer a ligação estável e fiável entre os 2 computadores, utilizando a porta de série. Para isso,

implementamos, tal como é referido no enunciado, as funções `llopen`, `llread`, `llwrite` e `llclose`.

*Consultar os anexos e para mais informação sobre o código utilizado*

## 5.1 LLOPEN

Esta função é responsável por estabelecer a ligação entre o emissor e o recetor através da porta de série.

Do lado do transmissor, esta função instala o alarme que vai ser utilizado ao longo da ligação, envia uma trama *SET* ao recetor, ficando depois à espera que este envie na resposta uma trama do tipo *UA*. Caso o recetor não responda passados 3 segundos, o emissor volta a reenviar a trama *SET*, aguardando depois uma resposta do outro lado. Caso volte a ficar sem resposta ao fim dos 3 segundos, repete o envio mais uma vez e no caso de mais um insucesso o programa termina. Caso o recetor responda com a trama *UA*, então a ligação é estabelecida.

Do lado do recetor, este aguarda o envio da trama *SET* por parte do emissor e responde com o envio de uma trama do tipo *UA*.

## 5.2 LLWRITE

A função `llwrite` é responsável pelo stuffing e envio das tramas do tipo I.

Inicialmente, é acrescentado um cabeçalho à mensagem, de acordo com o protocolo descrito no guião. De seguida, é feito o stuffing do *BCC2* e da mensagem, pelo que a trama está pronta para ser enviada.

O processo de envio das tramas do tipo I está protegido por um alarme com 3 segundos de espera e 3 tentativas.

Após o envio, é esperada uma resposta pela parte do recetor, através do comando *RR*, que simboliza que a trama foi transmitida corretamente, ou do comando *REJ*, que indica problemas no envio da trama, originando um reenvio da trama original.

## 5.3 LLREAD

A função `llread` recebe as tramas do tipo I enviadas pelo emissor.

A trama recebida é lida e analisada através de uma máquina de estados, sendo feitas as verificações do cabeçalho e do campo de dados e realizado o respetivo destuffing caso seja necessário.

Caso a trama recebida se trate de uma nova trama e não tenha erros no cabeçalho, mas possua erros no campo de dados, é enviada uma resposta

do tipo *REJ* para o emissor, pedindo uma retransmissão dessa trama. Caso contrário, é enviada uma resposta do tipo *RR*.

Se a trama recebida não possuir erros no cabeçalho e no campo de dados, ou caso seja um duplicado, é confirmada ao emissor através de uma trama *RR*.

Tramas com o cabeçalho errado são ignoradas, sem qualquer ação.

## 5.4 LLCLOSE

A função *llclose* tem como objetivo concluir a ligação entre o emissor e o recetor.

O emissor envia uma trama *DISC*, esperando por uma resposta do emissor da mesma trama *DISC*. Caso a receba, envia uma trama *UA* para finalizar a ligação.

O emissor está protegido por um alarme de 3 tentativas de 3 segundos de espera, tal como as funções mencionadas anteriormente.

O recetor espera por uma trama *DISC* e, caso a receba, envia de volta uma trama *DISC*, esperando por uma trama *UA* para finalizar a ligação.

## 6 Protocolo De Aplicação

O protocolo de aplicação permite a leitura de informação do ficheiro a enviar, fragmentando o ficheiro em tramas e preenchendo-o com um cabeçalho de controlo, sendo também capaz de descodificar a própria trama enviada.

**Consultar os anexos e para mais informação sobre o código utilizado**

Para que tal fosse implementado, recorremos às seguintes funções:

**OpenFile** Abre o ficheiro recebido e retorna os dados do ficheiro, assim como o seu tamanho.

**ParseControlPacket** Gera um pacote de controlo do tipo *START* ou *END*, contendo o tamanho e o nome do ficheiro.

**ParseDataPacket** Gera um pacote de dados, preenchendo-o com um cabeçalho contendo uma *FLAG* de controlo, o número de pacotes, o tamanho do ficheiro e o respetivo fragmento do ficheiro a ser enviado.

**SplitPacket** Divide o ficheiro em fragmentos mais pequenos.

**CheckStart** Verifica se o pacote de controlo foi recebido corretamente e obtém deste o tamanho e o nome do ficheiro.

**CheckEND** Compara o pacote de controlo do tipo *START* enviado antes da transmissão dos dados com o do tipo *END* recebido no final da transmissão, verificando se os campos com o tamanho e nome do ficheiro são iguais.

**AssembleDataPacket** Retorna o campo de dados de um pacote.

**CreateFile** Gera um ficheiro de acordo com os dados recebidos.

## 7 Validação

Foi testado o envio de vários ficheiros, incluindo ficheiros com uma elevada quantidade de dados, os quais foram enviados do emissor para o recetor corretamente, sem perda de informação.

Relativamente aos testes relativos à interrupção da ligação do cabo de série e geração de ruído, não fomos capazes de apresentar imagens relativas ao seu procedimento, porém, o seu sucesso foi comprovado na presença do docente no decurso da apresentação do projeto.

Para fortalecer ainda mais esta validação, recorremos ao envio de ficheiros simulando a ocorrência de erros no *BCC* e no *BCC2* com variação na percentagem de erros, à variação do tamanho dos pacotes, à variação das capacidades de ligação (*baudrate*) e à geração de atraso de propagação simulado. Os resultados obtidos são a seguir apresentados.

## 8 Eficiência Do Protocolo De Ligação De Dados

### 8.1 Variação da capacidade de Baudrate

Foi utilizada a imagem do pinguim (pinguim.png), com um tamanho de 35.4KB, sobre a qual se fez variar os valores do *baudrate*.

Foi possível concluir que o aumento do *baudrate* provoca uma diminuição da eficiência, embora o tempo de execução seja menor.

### 8.2 Variação do tamanho das tramas

Utilizando uma imagem de tamanho 35.4KB e um *baudrate* de 38400, fez-se variar o tamanho de envio das tramas em cada *llwrite*.

Foi possível concluir que o aumento do tamanho das tramas de envio provocou o aumento da eficiência, sendo o tempo de execução menor.



### 8.3 Atraso no envio das tramas

Utilizando uma imagem de tamanho 35.4KB, um *baudrate* de 38400 e o envio de 128 bytes em cada trama, introduziu-se um atraso no envio de cada trama no `llwrite`, através da função `usleep()`.

Tal como esperado, foi possível concluir que a introdução de um atraso no envio de tramas causa uma diminuição da eficiência do código, sendo o tempo de execução cada vez menor à medida que o atraso introduzido aumenta.

### 8.4 Geração de erros no cabeçalho e no campo de dados

Foram criadas duas funções, `generateRandomBCC` e `generateRandomBCC2`, de modo a gerar erros no cabeçalho e campo de dados, respetivamente, a uma percentagem definida no ficheiro `macros.h`, através das macros `BCC1ERRORRATE` e `BCC2ERRORRATE`.

Utilizando uma imagem de tamanho 35.4KB, um *baudrate* de 38400 e o envio de 128 bytes em cada trama, fez-se variar os valores de `BCC1ERRORRATE` e `BCC2ERRORRATE`.

Tal como esperado, foi possível concluir que o aumento da taxa de erros gerados no cabeçalho e campo de dados provocou uma diminuição da eficiência, também como um aumento do tempo de execução.

## 9 Conclusões

Em suma, foi possível alcançar o objetivo proposto do trabalho, a implementação de um protocolo de ligação de dados fiável através de uma porta série, sendo possível enviar com sucesso ficheiros de diferentes tamanhos e extensões. O projeto encontra-se dividido em duas camadas distintas: camada de ligação e camada de aplicação, tal como fomos explicando ao longo deste relatório.

Através deste trabalho, foi possível compreender não só o processo de implementação de um protocolo de ligação de dados, mas também as condições que afetam a eficiência do protocolo, através da alteração do tamanho da trama de envio, do *baudrate*, da quantidade de erros e do atraso no envio das tramas.

## 10 Anexos

### 10.1 Código

#### 10.2 emissor.h

```
1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <fcntl.h>
4 #include <termios.h>
5 #include <stdio.h>
6
7 #include <errno.h>
8 #include <signal.h>
9 #include <stdlib.h>
10 #include <time.h>
11
12 #include <string.h>
13
14 #include "llfunctions.h"
15 #include "application.h"
16
17
18 /**
19  * \brief main function that starts the program flow
20  * @param argc argument count
21  * @param argv char pointer array with the arguments
22  */
23 int main(int argc, char** argv);
```

### 10.3 emissor.c

```
1  /*Non-Canonical Input Processing*/
2  #include "emissor.h"
3
4  extern unsigned int packetNumber;
5
6  int main(int argc, char** argv)
7  {
8      int fd;
9
10     if ((argc < 3) || ((strcmp("/dev/ttyS0", argv[1])!=0) && (
11         strcmp("/dev/ttyS1", argv[1])!=0))) {
12         printf("Usage:\tnserial SerialPort File path\n\tex: nserial
13         /dev/ttyS1 \t filename.jpg \n");
14         return -1;
15     }
16
17     /*
18     Open serial port device for reading and writing and not as
19     controlling tty
20     because we don't want to get killed if linenoise sends CTRL-C.
21     */
22
23     struct timespec initialTime, finalTime;
24     clock_gettime(CLOCK_REALTIME, &initialTime);
25
26     if ((fd = open(argv[1], ORDWR | ONOCTTY)) < 0) {
27         perror(argv[1]);
28         return -2;
29     }
30
31     int fileNameSize = strlen(argv[2]);
32     char* filename = (char*)malloc(fileNameSize);
33     filename = (char*)argv[2];
34     off_t fileSize = 0;
35     int sizeControlPacket = 0;
36
37     unsigned char *data = openFile(filename, &fileSize);
38
39     // Dealing with the SET and UA
40     if (llopen(fd, TRANSMITTER) == ERROR){
41         puts("TRANSMITTER: Error on llopen");
42         return -3;
43     }
44
45     // Start Control packet
46     unsigned char *start = parseControlPacket(CT_START, fileSize,
```

```

45     filename , fileNameSize , &sizeControlPacket);
46     if(llwrite(fd, start , sizeControlPacket) != TRUE ){
47         puts("TRANSMITTER: Error writing START control packet");
48         return -4;
49     }
50     free(start);
51
52     // Ciclo de envio dos packets
53     int packetSize = PACKETSZ;
54     off_t index = 0;
55
56     while(index < fileSize && packetSize == PACKETSZ){
57         unsigned char* packet = splitPacket(data, &index, &
58             packetSize, fileSize);
59
60         int length = packetSize;
61
62         unsigned char* message = parseDataPacket(packet, fileSize ,
63             &length);
64
65         if(llwrite(fd, message, length) != TRUE){
66             puts("TRANSMITTER: Error sending data packet");
67             return -5;
68         }
69
70         printf("Sent packet number: %d\n", packetNumber);
71
72         free(message);
73     }
74
75     // End Control packet
76     unsigned char *end = parseControlPacket(CT_END, fileSize ,
77         filename , fileNameSize , &sizeControlPacket);
78
79     if(llwrite(fd, end, sizeControlPacket) != TRUE ){
80         puts("TRANSMITTER: Error writing END control packet");
81         return -6;
82     }
83     free(end);
84
85     if(llclose(fd, TRANSMITTER) == ERROR){
86         puts("TRANSMITTER: Error on llclose");
87         return -7;
88     }
89     clock_gettime(CLOCK_REALTIME, &finalTime);

```

```
90
91     double accum = (finalTime.tv_sec - initialTime.tv_sec) + (
92         finalTime.tv_nsec - initialTime.tv_nsec) / 1E9;
93
94     printf("Seconds passed: %f\n", accum);
95
96     sleep(1);
97     close(fd);
98     free(data);
99     return 0;
100 }
```

## 10.4 recetor.h

```
1  /*Non-Canonical Input Processing*/
2
3  #include <sys/types.h>
4  #include <sys/stat.h>
5  #include <fcntl.h>
6  #include <termios.h>
7  #include <stdio.h>
8  #include <string.h>
9
10 #include <time.h>
11
12
13 // Created files
14
15 #include "llfunctions.h"
16 #include "application.h"
```

## 10.5 recetor.c

```
1 #include "recetor.h"
2
3 extern unsigned int packetNumber;
4
5 int main(int argc, char** argv)
6 {
7     int fd;
8     off_t index = 0;
9
10    if ( (argc < 2) ||
11        ((strcmp("/dev/ttyS0", argv[1])!=0) &&
12         (strcmp("/dev/ttyS1", argv[1])!=0) )) {
13        printf("Usage:\tnserial SerialPort\n\tex: nserial /dev/ttyS1\n");
14        return -1;
15    }
16
17    /*
18     Open serial port device for reading and writing and not as
19     controlling tty
20     because we don't want to get killed if linenoise sends CTRL-C.
21 */
22    struct timespec initialTime, finalTime;
23    clock_gettime(CLOCK_REALTIME, &initialTime);
24
25    fd = open(argv[1], ORDWR | O_NOCTTY );
26    if (fd < 0) {
27        perror(argv[1]);
28        return -2;
29    }
30
31    if(llopen(fd, RECEIVER) == ERROR){
32        puts("Error on llopen");
33        return -3;
34    }
35
36    unsigned char* start = malloc(0);
37    unsigned int size, sizeStart;
38
39
40    size = llread(fd, start);
41    sizeStart = size;
42
43
44    unsigned int fileSize = 0;
45    unsigned int nameSize = 0;
```

```

46 char *fileName = (char *)malloc(0);
47
48 if(checkStart(start,&fileSize ,fileName,&nameSize) == ERROR){
49     puts("Error on checkStart");
50     return -4;
51 }
52
53 // Loop for reading all llwrites from the emissor
54 unsigned char* dataPacket;
55 unsigned int packetsRead = 0;
56 unsigned int messageSize;
57
58 unsigned char* final;
59
60 unsigned char* result = (unsigned char*)malloc(fileSize); //
    Creates null pointer to allow realloc
61
62 while(TRUE){
63     unsigned int packetSize = 0;
64     unsigned char* message = malloc(0);
65     messageSize = 0;
66
67     if((messageSize = llread(fd,message)) == ERROR){
68         puts("Error on llread data packet ");
69         return -5;
70     }
71     printf("message size = %d\n",messageSize);
72
73     if(messageSize == 0){
74         continue;
75     }
76     else if (message[0] == CT_END){
77         puts("Reached Control End Packet");
78         final = (unsigned char*)malloc(messageSize);
79         memcpy(final ,message ,messageSize);
80         break;
81     }
82
83     packetsRead++;
84
85     printf("Received packet number: %d\n", packetsRead);
86
87     dataPacket = assembleDataPacket(message ,messageSize ,&
    packetSize);
88
89     for(int i= 0; i < packetSize; i++){
90         result[index + i] = dataPacket[i];
91     }
92

```



```

93     index += packetSize;
94
95     free(dataPacket);
96 }
97
98 if(checkEND(start, sizeStart, final, messageSize) == 1) {
99     puts("Start and End packets are different!");
100     return -6;
101 }
102
103 printf("Received a file of size: %u\n", fileSize);
104
105 // Creating the file to be rewritten after protocol
106 createFile(result, fileSize, fileName);
107
108
109 if(llclose(fd, RECEIVER) == ERROR){
110     puts("Error on llclose");
111     return -7;
112 }
113
114
115 clock_gettime(CLOCK_REALTIME, &finalTime);
116 double accum = (finalTime.tv_sec - initialTime.tv_sec) + (
117     finalTime.tv_nsec - initialTime.tv_nsec) / 1E9;
118 printf("Seconds passed: %f\n", accum);
119 sleep(1);
120
121 free(fileName);
122 free(result);
123
124 close(fd);
125
126 return 0;
127 }

```

## 10.6 llfunctions.h

```
1
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <unistd.h>
5 #include <sys/types.h>
6 #include <sys/stat.h>
7 #include <fcntl.h>
8 #include <termios.h>
9 #include <errno.h>
10 #include <signal.h>
11 #include <string.h>
12
13 #include "stateMachines.h"
14 #include "macros.h"
15
16 /**
17  * \brief Deals with the protocol initiation establishment
18  * @param fd file descriptor for the serial port to be used for
19  * the connection
20  * @param status If 0, sends SET message and waits for UA, if 1,
21  * waits for set and sends UA
22  * @return returns 0 upon success, -1 otherwise
23  */
24 int llopen(int fd, int status);
25
26 /**
27  * \brief gets BCC2
28  * @param message gets BCC2 from this message
29  * @param size message size
30  * @return returns BCC2
31  */
32 unsigned char getBCC2(unsigned char *message, int size);
33
34 /**
35  * \brief stuffs BCC2
36  * @param bcc2 bcc2 char to be stuffed
37  * @param size size of BCC2 after stuffing
38  * @return returns the stuffed BCC2
39  */
40 unsigned char* stuffBCC2(unsigned char bcc2, unsigned int *size);
41
42 /**
43  * \brief Sends an I packet from a message from buffer to the
44  * serial port
45  * @param fd file descriptor of the serial port
46  * @param buffer containing the message to be sent
```

```

45  * @param length length of the message to be sent
46  * @return TRUE(1) upon sucess, FALSE(0) upon failure
47  */
48  int llwrite(int fd, unsigned char *buffer, int length);
49
50  /**
51   * \brief Reads an I packets sent trough the serial port
52   * @param fd file descriptor for the serial port
53   * @param buffer buffer read from the serial port
54   * @return size of the read buffer
55   */
56  unsigned int llread(int fd, unsigned char *buffer);
57
58  /**
59   * \brief Termination of the protocol by serial port
60   * @param fd file descriptor of the serial port
61   * @param status if 0, acts as sender. if 1, acts as receiver
62   *               for the termination protocol
63   * @return returns 0 upon sucess, -1 otherwise
64   */
65  int llclose(int fd, int status);
66
67  /**
68   * \brief handles the alarm
69   * @param signo signal number to be handled
70   */
71  void alarmHandler(int signo);
72
73  unsigned char* generateRandomBCC(unsigned char* packet, int
    packetSize);
74
75  unsigned char* generateRandomBCC2(unsigned char* packet, int
    packetSize);

```

## 10.7 llfunctions.c

```
1 #include "llfunctions.h"
2
3
4 struct termios oldtio, newtio;
5
6 volatile int STP=FALSE;
7 extern unsigned char rcv;
8 int counter = 0;
9 int trama = 0;
10 extern int res;
11
12
13 int llopen(int fd, int status) {
14
15     if (tcgetattr(fd, &oldtio) == -1) { /* save current port
16 settings */
17         perror("llopen: tcgetattr");
18         return ERROR;
19     }
20
21     bzero(&newtio, sizeof(newtio));
22     newtio.c_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;
23     newtio.c_iflag = IGNPAR;
24     newtio.c_oflag = 0;
25
26     /* set input mode (non-canonical, no echo,...) */
27     newtio.c_lflag = 0;
28
29     newtio.c_cc[VTIME]      = 20; /* inter-character timer
30 unused */
31     newtio.c_cc[VMIN]       = 0; /* blocking read until 0 chars
32 received */
33
34     /*
35     VTIME e VMIN devem ser alterados de forma a proteger com um
36 temporizador a
37 leitura do(s) proximo(s) caracter(es)
38 */
39
40     tcflush(fd, TCIOFLUSH);
41
42     if (tcsetattr(fd, TCSANOW, &newtio) == -1) {
43         perror("tcsetattr");
44         return ERROR;
45     }
46
47     puts("New termios structure set");
48 }
```

```

44
45
46     if(status == TRANSMITTER) {
47
48         // Installing Alarm Handler
49         if(signal(SIGALRM, alarmHandler) || siginterrupt(SIGALRM
50 , 1)){
51             puts("Signal instalation failed");
52             return ERROR;
53         }
54
55         counter = 0;
56         do{
57             int wr;
58             if((wr = sendMessage(fd, C_SET)) != ERROR){
59                 printf("llopen: C_SET message sent: %d \n", wr);
60             }
61             else{
62                 puts("llopen: Error sending message");
63             }
64
65             alarm(TIMEOUT); // Call an alarm to wait for the
66 message
67
68             if(receiveUA(fd) == TRUE){
69                 puts("TRANSMITTER: UA received\n");
70                 STP = TRUE;
71                 counter = 0;
72                 alarm(0);
73             }
74
75             }while(STP == FALSE && counter < MAXTRIES);
76
77         }
78         else if(status == RECEIVER) {
79             if(readSetMessage(fd) == TRUE) {
80                 puts("RECEIVER: Read SET message correctly");
81                 if(sendMessage(fd, C_UA) == -1) {
82                     fprintf(stderr, "llopen - Error writing to
83 serial port (Receiver)\n");
84                     return ERROR;
85                 }
86             }
87             else {
88                 puts("RECEIVER: Sent UA message");
89             }
90
91         }
92         else {
93             fprintf(stderr, "llopen - Error reading from serial
94 port (Receiver)\n");
95             return ERROR;
96         }
97     }
98 }

```

```

89     }
90 }
91 return 0;
92 }
93
94 unsigned char getBCC2(unsigned char *mensagem, int size){
95
96     unsigned char bcc2 = mensagem[0];
97
98     for(int i = 1; i < size; i++){
99         bcc2 ^= mensagem[i];
100     }
101     return bcc2;
102 }
103
104 unsigned char* stuffBCC2(unsigned char bcc2, unsigned int *size)
105 {
106     unsigned char* stuffed;
107     if(bcc2 == FLAG){
108         stuffed = malloc(2 * sizeof(unsigned char));
109         stuffed[0] = ESCAPE.BYTE;
110         stuffed[1] = ESCAPE.FLAG;
111         (*size) = 2;
112     }
113     else if(bcc2 == ESCAPE.BYTE){
114         stuffed = malloc(2 * sizeof(unsigned char));
115         stuffed[0] = ESCAPE.BYTE;
116         stuffed[1] = ESCAPE.ESCAPE;
117         (*size) = 2;
118     }
119     else{
120         stuffed = malloc(sizeof(unsigned char));
121         stuffed[0] = bcc2;
122         (*size) = 1;
123     }
124     return stuffed;
125 }
126
127 int llwrite(int fd, unsigned char *buffer, int length) {
128 // escreve a trama e fica a espera de receber uma mensagem RR ou
129 // REJ para saber o que enviar a seguir
130     unsigned char bcc2;
131     unsigned int sizebcc2 = 1;
132     unsigned int messageSize = length+6;
133     unsigned char *bcc2Stuffed = (unsigned char *)malloc(sizeof(
134     unsigned char));
135     unsigned char *message = (unsigned char *)malloc(messageSize
136     * sizeof(unsigned char));

```

```

134
135     bcc2 = getBCC2(buffer , length);
136     bcc2Stuffed = stuffBCC2(bcc2, &sizebcc2);
137
138
139     // Inicio do preenchimento da mensagem
140     message[0] = FLAG;
141     message[1] = A.EE;
142     if(trama == 0){
143         message[2] = NS0;
144     }
145     else{
146         message[2] = NS1;
147     }
148     message[3] = message[1] ^ message[2];
149
150     // Começa a ler do 4 e o tamanho depende da mensagem a ser
151     enviada
152     int i = 4;
153     for(int j = 0; j < length; j++){
154         if(buffer[j] == FLAG){
155             message = (unsigned char *)realloc(message, ++
156             messageSize);
157             message[i] = ESCAPE.BYTE;
158             message[i + 1] = ESCAPE.FLAG;
159             i+=2;
160         }
161         else if(buffer[j] == ESCAPE.BYTE){
162             message = (unsigned char *)realloc(message, ++
163             messageSize);
164             message[i] = ESCAPE.BYTE;
165             message[i+1] = ESCAPE.ESCAPE;
166             i+=2;
167         }
168         else{
169             message[i] = buffer[j];
170             i++;
171         }
172     }
173
174     if(sizebcc2 == 2){
175         message = (unsigned char *)realloc(message, ++
176         messageSize);
177         message[i] = bcc2Stuffed[0];
178         message[i + 1] = bcc2Stuffed[1];
179         i+=2;
180     }
181     else{
182         message[i] = bcc2;

```

```

179         i++;
180     }
181     message[i] = FLAG;
182
183     //Mensagem preenchida Trama I feita
184     // printMessage
185
186     for(int j = 0; j < messageSize; j++){
187         printf("message[%d] = 0x%X\n", j, message[j]);
188     }
189
190
191     counter = 0;
192     STP = FALSE;
193
194     // Envio da trama
195     do {
196         // Processo de escrita
197         // tcflush(fd,TCIOFLUSH);
198
199         unsigned char* copyBcc = (unsigned char *)malloc(
messageSize);
200         unsigned char* copyBcc2 = (unsigned char *)malloc(
messageSize);
201
202         copyBcc = generateRandomBCC(message, messageSize);
203         copyBcc2 = generateRandomBCC2(copyBcc, messageSize);
204
205
206         int wr = write(fd, copyBcc2, messageSize);
207
208         printf("TRANSMITTER: SET message sent: %d bytes sent\n",
wr);
209
210         alarm(TIMEOUT);
211
212         readReceiverMessage(fd);
213
214         // Tratar do rcv
215         if((rcv == RR0 && trama == 1) || (rcv == RR1 && trama ==
0)) {
216             counter = 0;
217             trama = (trama + 1) % 2;
218             STP = FALSE;
219             alarm(0);
220             if(rcv == RR0) {
221                 puts("TRANSMITTER: Received RR0");
222             }
223             else {

```



```

224         puts("TRANSMITTER: Received RRI");
225     }
226     break;
227 }
228
229     else if(rcv == REJ0 || rcv == REJ1) {
230         STP = TRUE;
231         //alarm(0);
232         if(rcv == REJ0) {
233             puts("TRANSMITTER: Received REJ0");
234         }
235         else {
236             puts("TRANSMITTER: Received REJ1");
237         }
238     }
239
240     else if(res == 0) {
241         puts("TRANSMITTER: Don't read any message from
Receiver");
242         STP = TRUE;
243     }
244
245     else {
246         puts("TRANSMITTER: Received an invalid message");
247     }
248
249 } while(STP && counter < MAXTRIES);
250
251 if(counter >= MAXTRIES) {
252     return FALSE;
253 }
254 else {
255     return TRUE;
256 }
257 }
258
259 unsigned int llread(int fd, unsigned char* buffer) {
260
261     unsigned int size = 0;
262     receiverRead_StateMachine(fd,buffer , &size);
263
264     printf("size llread = %d\n",size);
265
266     return size;
267 }
268
269 int llclose(int fd, int status) {
270     //emissor:
271     // envia DISC, espera porInteraction received\n

```

```

272     if(status == TRANSMITTER) {
273
274         counter = 0;
275         STP = FALSE;
276
277         do {
278             int wr;
279             if((wr = sendMessage(fd, C_DISC)) != ERROR){
280                 puts("TRANSMITTER: C_DISC message sent");
281             }
282             else{
283                 puts("TRANSMITTER: Error sending C_DISC message"
284 );
285             }
286
287             alarm(TIMEOUT); // Call an alarm to wait for the
288 message
289
290             if(receiveDISC(fd) == 0 && res != 0){
291                 puts("TRANSMITTER: C_DISC received");
292                 STP = TRUE;
293                 counter = 0;
294                 alarm(0);
295             }
296         } while(STP == FALSE && counter < MAXTRIES);
297
298         if(sendMessage(fd, C_UA)) {
299             puts("TRANSMITTER: Send UA");
300         }
301         tcsetattr(fd, TCSANOW, &oldtio);
302     }
303 //recetor:
304 // le a mensagem DISC enviada pelo emissor, envia DISC e recebe
305 UA
306     else if(status == RECEIVER) {
307         if (receiveDISC(fd) == 0) {
308             puts("RECEIVER: Read DISC");
309             if(sendMessage(fd, C_DISC)) {
310                 puts("RECEIVER: Send DISC");
311                 if(receiveUA(fd) == TRUE) {
312                     puts("RECEIVER: Read UA");
313                 }
314             }
315             else {
316                 fprintf(stderr, "llclose- Error reading UA
317 message (Receiver)\n");
318                 return ERROR;
319             }
320         }

```

```

317     }
318
319     else {
320         fprintf(stderr, "llclose- Error writing DISC
message to serial port (Receiver)\n");
321         return ERROR;
322     }
323 }
324
325     else {
326         fprintf(stderr, "llclose - Error reading DISC
message (Receiver)\n");
327         return ERROR;
328     }
329     tcsetattr(fd, TCSANOW, &oldtio);
330 }
331 return 0;
332 }
333
334
335
336 void alarmHandler(int signo){
337
338     counter++;
339     if(counter >= MAXTRIES){
340         printf("Exceeded maximum amount of tries: (%d)\n", MAXTRIES)
;
341         exit(0);
342     }
343     return ;
344 }
345
346
347
348 unsigned char* generateRandomBCC(unsigned char* packet, int
packetSize){
349     unsigned char* copy = (unsigned char *)malloc(packetSize);
350     memcpy(copy, packet, packetSize);
351
352     if(((rand() % 100) + 1) <= BCCIERRORRATE){
353         unsigned char hex[16] = {'0', '1', '2', '3', '4', '5', '6', '7',
'8', '9', 'A', 'B', 'C', 'D', 'E', 'F'};
354
355         copy[(rand() % 3) + 1] = hex[rand() % 16];
356         puts("BCC Value sucessfully changed");
357     }
358     return copy;
359 }
360

```

```

361
362 unsigned char* generateRandomBCC2(unsigned char* packet, int
    packetSize){
363     unsigned char* copy = (unsigned char *)malloc(packetSize);
364     memcpy(copy, packet, packetSize);
365
366     if(((rand() % 100) + 1) <= BCC2ERRORRATE){
367         unsigned char hex[16] = {'0','1','2','3','4','5','6','7',
            '8','9','A','B','C','D','E','F'};
368
369         copy[(rand() % (packetSize - 5)) + 4] = hex[rand() %
            16];
370         puts("BCC2 Value sucessfully changed");
371     }
372     return copy;
373 }

```

## 10.8 stateMachines.h

```
1 #include <stdlib.h>
2 #include <termios.h>
3 #include <stdio.h>
4 #include <errno.h>
5 #include <unistd.h>
6
7
8 #include "macros.h"
9
10
11 enum state {
12     START,
13     FLAG_RCV,
14     A_RCV,
15     C_RCV,
16     BCC_OK,
17     BYTE_DESTUFFING,
18     STOP
19 };
20
21 int sendMessage(int fd, unsigned char c);
22
23 int readSetMessage(int fd);
24
25 int readReceiverMessage(int fd);
26
27 int receiveUA(int serialPort);
28
29 int receiverRead_StateMachine(int fd, unsigned char* frame ,
30     unsigned int *size);
31
32 int receiveDISC(int fd);
33
34 int checkBCC2(unsigned char *packet, int size);
```

## 10.9 stateMachines.c

```
1 #include "stateMachines.h"
2
3 unsigned char rcv;
4 int expectedTrama = 0;
5 int res;
6 int counter_errors = 0;
7
8
9 int sendMessage(int fd, unsigned char c){
10
11     unsigned char message[5];
12
13     message[0] = FLAG;
14     message[1] = A_EE;
15     message[2] = c;
16     message[3] = A_EE ^ c;
17     message[4] = FLAG;
18
19     return write(fd, message, 5);
20 }
21
22
23
24 int readSetMessage(int fd) {
25     enum state current = START;
26
27     int finish = FALSE;
28     unsigned char r;
29
30     while (finish == FALSE){
31         res = read(fd, &r, 1);
32
33         switch (current){
34             case START:
35                 if (r == FLAG){
36                     current = FLAG_RCV;
37                 }
38                 break;
39             case FLAG_RCV:
40                 if (r == A_EE){
41                     current = A_RCV;
42                 }
43                 else if (r == FLAG){
44                     current = FLAG_RCV;
45                 }
46                 else{
47                     current = START;
```

```

48     }
49     break;
50     case A_RCV:
51         if (r == C_SET){
52             current = C_RCV;
53         }
54         else if (r == FLAG){
55             current = FLAG_RCV;
56         }
57         else{
58             current = START;
59         }
60         break;
61     case C_RCV:
62         if (r == (C_SET ^ A.EE)){
63             current = BCC_OK;
64         }
65         else if (r == FLAG){
66             current = FLAG_RCV;
67         }
68         else{
69             current = START;
70         }
71         break;
72     case BCC_OK:
73         if (r == FLAG){
74             finish = TRUE;
75         }
76         else{
77             current = START;
78         }
79         break;
80     default:
81         break;
82     }
83 }
84 return finish;
85 }
86
87 int readReceiverMessage(int fd) {
88     enum state current = START;
89
90     int finish = FALSE;
91     unsigned char r, check;
92
93     while (finish == FALSE){
94         res = read(fd, &r, 1);
95
96         if(res == 0) {

```

```

97         finish = TRUE;
98     }
99
100     switch (current){
101     case START:
102         if (r == FLAG){
103             current = FLAG.RCV;
104         }
105         break;
106     case FLAG.RCV:
107         if (r == A.EE){
108             current = A.RCV;
109         }
110         else if (r == FLAG){
111             current = FLAG.RCV;
112         }
113         else{
114             current = START;
115         }
116         break;
117     case A.RCV:
118         if (r == REJ0 || r == REJ1 || r == RR0 || r == RR1 )
119     {
120         current = C.RCV;
121         check = r;
122         rcv = r;
123     }
124     else if (r == FLAG){
125         current = FLAG.RCV;
126     }
127     else{
128         current = START;
129     }
130     break;
131 case C.RCV:
132     if (r == (check ^ A.EE)){
133         current = BCC.OK;
134     }
135     else if (r == FLAG){
136         current = FLAG.RCV;
137     }
138     else{
139         current = START;
140     }
141     break;
142 case BCC.OK:
143     if (r == FLAG){
144         finish = TRUE;
145     }

```



```

145         else{
146             current = START;
147         }
148         break;
149     default:
150         break;
151     }
152 }
153 return finish;
154 }
155
156 int receiveUA(int fd){
157     unsigned char c; // char read. Changes the state
158     int nr;
159     enum state current = START;
160
161     int STP = FALSE;
162     while(STP == FALSE){
163         nr = read(fd,&c,1);
164
165         if (nr < 0) {
166             if (errno == EINTR) {
167                 puts("Timed out. Sending again.");
168                 return ERROR;
169             }
170             continue;
171         }
172
173         //State Machine
174         switch(current){
175             case START:
176                 if(c == FLAG){
177                     current = FLAG.RCV;
178                 }
179                 break;
180             case FLAG.RCV:
181                 if(c == A_EE){
182                     current = A.RCV;
183                 }
184                 else if(c == FLAG) {
185                     current = FLAG.RCV;
186                 }
187                 else{
188                     current = START;
189                 }
190                 break;
191             case A.RCV:
192                 if(c == C_UA){
193                     current = C.RCV;

```

```

194         }
195         else if (c == FLAG){
196             current = FLAG_RCV;
197         }
198         else {
199             current = START;
200         }
201         break;
202     case C_RCV:
203         if(c == (C_UA ^ A_EE)){
204             current = BCC.OK;
205         }
206         else if( c == FLAG){
207             current = FLAG_RCV;
208         }
209         else{
210             current = START;
211         }
212         break;
213     case BCC.OK:
214         if(c == FLAG){
215             current = STOP;
216             STP = TRUE;
217         }
218         else{
219             current = START;
220         }
221         break;
222     case STOP:
223         break;
224     default:
225         break;
226     }
227 }
228
229 //puts("exiting state machine");
230 return TRUE;
231 }
232 }
233
234
235 int receiverRead_StateMachine(int fd, unsigned char* frame,
236                               unsigned int *size) {
237
238     unsigned char buf, check = 0;
239     int trama = 0;
240     enum state current = START;
241     int correctBCC2 = FALSE; // if no errors in BCC2,
242     correctBCC2 = 1; else correctBCC2 = 0

```

```

241     int errorOnDestuffing = FALSE; // if no errors occur on
242     destuffing, the var stays equal to 0, else the value is 1
243     counter_errors++;
244     puts("Receiver reading frames");
245     while(current != STOP) {
246
247         res = read(fd, &buf, 1);
248         printf("read : 0x%X\n", buf);
249
250         if(res == ERROR) {
251             fprintf(stderr, "llread() - Error reading from
buffer");
252             return -1;
253         }
254
255         switch (current)
256         {
257             case START:
258                 if(buf == FLAG) {
259                     current = FLAG.RCV;
260                 }
261                 break;
262
263             case FLAG.RCV:
264                 if(buf == A.EE) {
265                     current = A.RCV;
266                 }
267                 else if(buf != FLAG) {
268                     current = START;
269                 }
270                 break;
271
272             case A.RCV:
273                 if(buf == NS0) {
274                     current = C.RCV;
275                     check = buf;
276                     trama = 0;
277                 }
278
279                 else if(buf == NS1) {
280                     current = C.RCV;
281                     check = buf;
282                     trama = 1;
283                 }
284
285                 else if(buf == FLAG) {
286                     current = FLAG.RCV;
287                 }

```

```

288
289         else {
290             current = START;
291         }
292         break;
293
294     case C_RCV:
295         if(buf == (A_EE ^ check)) {
296             current = BCC_OK;
297         }
298
299         else if(buf == FLAG) {
300             current = FLAG_RCV;
301         }
302
303         else {
304             current = START;
305         }
306         break;
307
308     case BCC_OK:
309         if(buf == FLAG) {
310             if(checkBCC2(frame, *size) != ERROR) {
311                 correctBCC2 = TRUE;
312                 current = STOP;
313             }
314             else {
315                 correctBCC2 = FALSE;
316                 current = STOP;
317             }
318         }
319         else if(buf == ESCAPE_BYTE) {
320             current = BYTE_DESTUFFING;
321         }
322         else {
323             (*size)++;
324             frame = (unsigned char *)realloc(frame, *size);
325             frame[*size - 1] = buf; // still receiving data
326         }
327     }
328
329     break;
330
331     case BYTE_DESTUFFING:
332         if(buf == ESCAPE_FLAG) {
333             frame = (unsigned char *)realloc(frame, ++(*size));
334             frame[*size - 1] = FLAG;
335         }

```

```

336     }
337
338     else if(buf == ESCAPEESCAPE) {
339         frame = (unsigned char *)realloc(frame, ++(*size
340     ));
341         frame[*size - 1] = ESCAPE_BYTE;
342     }
343
344     else {
345         puts("Character after escape character not
346     recognized"); //can occur if there is an interference
347         errorOnDestuffing = TRUE;
348     }
349
350     current = BCC_OK;
351     break;
352
353     default:
354     break;
355 }
356
357 printf("total size: %d\n", *size);
358 frame = (unsigned char *)realloc(frame, *size - 1);
359 *size = *size - 1;
360
361 printf("Expected trama: %i\n", expectedTrama);
362 printf("Received trama: %i\n", trama);
363
364 if(correctBCC2 && !errorOnDestuffing) {
365     if(trama == expectedTrama) {
366         if(trama == 0) {
367             //send RR(Nr = 1)
368             sendMessage(fd, RR1);
369             puts("Receiver send RR1");
370         }
371
372         else {
373             //send RR(Nr = 0)
374             sendMessage(fd, RR0);
375             puts("Receiver send RR0");
376         }
377
378         expectedTrama = (expectedTrama + 1) % 2;
379     }
380
381     else { //ao fazermos isto garantimos que a trama
382         recebida e repetida
383         *size = 0;

```

```

382
383         if(expectedTrama == 0) {
384             //send RR(Nr = 0)
385             sendMessage(fd, RR0);
386             puts("Receiver send RR0 after repeated
information");
387         }
388         else {
389             //send RR(Nr = 1)
390             sendMessage(fd, RR1);
391             puts("Receiver send RR1 after repeated
information");
392         }
393     }
394 }
395 else { //caso BCC2 tenha erros ou tenha havido
interferencias
396     if(trama != expectedTrama) {
397         if(trama == 0) {
398             //send RR(Nr = 1)
399             sendMessage(fd, RR1);
400             expectedTrama = 1;
401             puts("Receiver send RR1 after errors in BCC2");
402         }
403         else {
404             //send RR(Nr=0)
405             sendMessage(fd, RR0);
406             expectedTrama = 0;
407             puts("Receiver send RR0 after errors in BCC2");
408         }
409     }
410
411     else { //trama correta, mas com erro em BCC2
412         *size = 0;
413
414         if(trama == 0) {
415             //send REJ 0
416             sendMessage(fd, REJ0);
417             expectedTrama = 0;
418             puts("Receiver send REJ0");
419         }
420
421         else {
422             //send REJ1
423             sendMessage(fd, REJ1);
424             expectedTrama = 1;
425             puts("Receiver send REJ1");
426         }
427     }

```

```

428     }
429     return 0;
430 }
431
432 int receiveDISC(int fd) {
433     //tcflush(fd, TCIOFLUSH); //limpa informacao recebida mas
434     //nao lida e informacao escrita mas nao transmitida
435
436     enum state current = START;
437
438     int finish = FALSE;
439     unsigned char r;
440
441     while (finish == FALSE)
442     {
443         res = read(fd, &r, 1);
444
445         if(res == 0) {
446             finish = TRUE;
447         }
448
449         switch (current)
450         {
451             case START:
452                 if (r == FLAG){
453                     current = FLAG.RCV;
454                 }
455                 break;
456             case FLAG.RCV:
457                 if (r == A_EE){
458                     current = A.RCV;
459                 }
460                 else if (r == FLAG){
461                     current = FLAG.RCV;
462                 }
463                 else{
464                     current = START;
465                 }
466                 break;
467             case A.RCV:
468                 if (r == C_DISC){
469                     current = C.RCV;
470                 }
471                 else if (r == FLAG){
472                     current = FLAG.RCV;
473                 }
474                 else{
475                     current = START;

```

```

476         }
477         break;
478     case C_RCV:
479         if (r == (C_DISC ^ A_EE)){
480             current = BCC_OK;
481         }
482         else if (r == FLAG){
483             current = FLAG_RCV;
484         }
485         else{
486             current = START;
487         }
488         break;
489     case BCC_OK:
490         if (r == FLAG){
491             finish = TRUE;
492         }
493         else{
494             current = START;
495         }
496         break;
497     default:
498         break;
499     }
500 }
501 return 0;
502 }
503
504 int checkBCC2(unsigned char *packet, int size){
505     int i;
506     unsigned char byte = packet[0];
507
508     for(i = 1; i < size - 1; i++) {
509         byte ^= packet[i];
510     }
511
512     if(byte == packet[size - 1]) {
513         return TRUE;
514     }
515     else{
516         return ERROR;
517     }
518 }
519 }

```



## 10.10 application.h

```
1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <unistd.h>
4
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <string.h>
8
9 #include "macros.h"
10
11 /**
12  * \brief opens the file sent and returns its data and size
13  * @param filename file to be read
14  * @param fileSize returns size of the file after being read
15  * @return returns the data of the read file
16  */
17 unsigned char* openFile(char *filename, off_t *fileSize);
18
19
20 /**
21  * \brief Generates the control packet for a given file
22  * @param state FLAG to distinguish END from START packet. START
23  *   = 0x02, END = 0x03
24  * @param filesize size of the read file
25  * @param filename name of the read file
26  * @param sizeFileName size of the name of the read file
27  * @param sizeControlPacket returns the size of the
28  *   generated control packet
29  * @return returns the generated control packet
30  */
31 unsigned char* parseControlPacket(unsigned int state, off_t
32   fileSize, char* filename, int sizeFileName, int *
33   sizeControlPacket);
34
35 /**
36  * \brief codifies the Given message into a packet according to
37  *   the protocol
38  * @param message message to be sent
39  * @param fileSize total size of the file to be written
40  * @param length total size of the packet to be sent through
41  *   llwrite serial port
42  */
43 unsigned char* parseDataPacket(unsigned char *message, off_t
44   fileSize, int *length);
45
46 /**
```

```

41  * \brief Splits the data into packets that fit into a message (
    * currently set to 128 bytes)
42  * @param message message containing the whole data
43  * @param index index to start/continue to write the data from
44  * @param packetSize returns the ammount of bytes that can be
    * written in a single llwrite (128 or less if end of file
    * reached)
45  * @param fileSize file total size to check how many bytes
    * should be written
46  * @return returns the packet data that will be sent
47  */
48  unsigned char* splitPacket(unsigned char *message, off_t *index,
    int *packetSize, off_t fileSize);
49
50
51 /**
52  * \brief Checks if the first packet read from the sender is
    * indeed the control start packet
53  * @param start packet read (first packet)
54  * @param fileSize gets the total size of the file through the
    * control packet
55  * @param name gets the filename through the control packet
56  * @param nameSize gets the filename size through the control
    * packet
57  */
58  int checkStart(unsigned char* start, unsigned int *fileSize,
    char *name, unsigned int *nameSize);
59
60
61 /**
62  * \brief Checks if the control END packet is equal to the START
    * control packet
63  * @param start start packet read (first packet)
64  * @param startSize size of the start packet
65  * @param end end packet read (last packet)
66  * @param endSize size of the end packet
67  * @return 0 if both packets are equal, 1 otherwise
68  */
69  int checkEND(unsigned char *start, int startSize, unsigned char
    *end, int endSize);
70
71 /**
72  * \brief Creates a the packet to be sent trough llwrite serial
    * port
73  * @param message Received message to be sent
74  * @param messageSize size of the received message
75  * @param packetSize size of the created packet to be returned
76  * @return Assembled packet to be sent
77  */

```

```

78 unsigned char* assembleDataPacket(unsigned char* message,
    unsigned int messageSize, unsigned int *packetSize);
79
80 /**
81  * \brief Reads a file given a name and data
82  * @param data file data
83  * @param fileSize size of the file to be created
84  * @param filename name of the file to be created
85  */
86 void createFile(unsigned char* data, unsigned int fileSize, char
    *filename);

```

## 10.11 application.c

```
1 #include "application.h"
2
3 unsigned int packetNumber = 0; //Global variable counting the
   number of packets being sent
4
5
6
7 unsigned char* openFile(char* filename, off_t *fileSize){
8
9     FILE * file;
10    struct stat st;
11    unsigned char *data;
12
13    if((file = fopen(filename, "r")) == NULL ){
14        perror("Cannot open file");
15        exit(-1);
16    }
17
18
19    stat(filename, &st);
20
21    *fileSize = st.st_size;
22
23    printf("Read a file with size %ld bytes\n", *fileSize);
24
25    data = (unsigned char *)malloc(*fileSize);
26
27    fread(data, sizeof(unsigned char), *fileSize, file);
28
29    if(ferror(file)){
30        perror("Error reading file");
31        exit(-2);
32    }
33
34    if(fclose(file) == EOF){
35        perror("Cannot close file");
36        exit(-1);
37    }
38    return data;
39 }
40
41 unsigned char* parseControlPacket(unsigned int state, off_t
   fileSize, char* filename, int sizeFilename, int *
   sizeControlPacket){
42
43     *sizeControlPacket = 5 + sizeof(fileSize) + sizeFilename;
44 }
```

```

45     unsigned char* packet = (unsigned char* )malloc(sizeof(
46     unsigned char) * (*sizeControlPacket));
47
48     if(state == CT.START){
49         packet[0] = CT.START;
50     }
51     else{
52         packet[0] = CT.END;
53     }
54     packet[1] = T1;
55     packet[2] = sizeof( fileSize );
56
57     for(int i = 0; i < packet[2]; i++){
58         packet[3+i] = ( fileSize >> (i*8)) & 0xFF;
59     }
60
61     packet[3 + packet[2]] = T2;
62     packet[4 + packet[2]] = sizeFilename;
63
64     for (int i = 0; i < sizeFilename; i++ ){
65
66         packet[5 + packet[2] + i] = filename[i];
67     }
68
69     return packet;
70 }
71
72 unsigned char* parseDataPacket(unsigned char *message, off_t
73 fileSize , int *length){
74
75     unsigned char *packet = (unsigned char*)malloc(fileSize + 4)
76     ;
77
78     packet[0] = CONTROL;
79     packet[1] = packetNumber % 255;
80     packet[2] = fileSize / 256;
81     packet[3] = fileSize % 256;
82
83     // Preencher o packet atraves da mensagem
84     for(int i = 0; i < *length; i++){
85         packet[4 + i] = message[i];
86     }
87
88     *length += 4;
89     packetNumber++;
90
91     return packet;
92 }

```

```

91 unsigned char* splitPacket(unsigned char *packet, off_t *index,
    int *packetSize, off_t fileSize){
92
93     unsigned char *splitPacket;
94
95     if(*index + *packetSize > fileSize){
96         *packetSize = fileSize - *index;
97     }
98
99     splitPacket = (unsigned char*)malloc(*packetSize);
100
101     for(int i = 0; i < *packetSize; i++){
102         splitPacket[i] = packet[*index];
103         (*index)++;
104     }
105
106     return splitPacket;
107 }
108
109 int checkStart(unsigned char* start, unsigned int *filesize,
    char *name, unsigned int *nameSize){
110
111     int fileSizeBytes;
112
113     // Checking control flag
114     if(start[0] != CTSTART || start[1] != T1){
115         puts("checkStart: Error checking CT.START or T1 flags");
116         return -1;
117     }
118
119     fileSizeBytes = (int)start[2];
120
121     // Getting fileSize
122     for(int i = 0; i < fileSizeBytes; i++){
123         *filesize |= (start[3 + i] << (i*8));
124     }
125
126     if(start[fileSizeBytes + 3] != T2){
127         puts("checkSart: Error checking T2");
128         return -1;
129     }
130
131     // Getting nameSize
132     *nameSize = (unsigned int)start[fileSizeBytes + 4];
133
134     // Getting fileName
135     name = (char *)realloc(name, *nameSize);
136
137

```

```

138     for(int i = 0; i < *nameSize; i++){
139         name[i] = start[fileSizeBytes + 5 + i];
140     }
141
142     return 0;
143 }
144
145 int checkEND(unsigned char *start, int startSize, unsigned char
146 *end, int endSize) {
147     int j = 5;
148
149     if(startSize != endSize) {
150         puts("checkEND: Start and End packets have differente
151 sizes");
152         return 1;
153     }
154     else {
155         if(end[0] == CTEND) {
156             for(int i = 5; i < startSize; i++) {
157                 if(start[i] != end[j]) {
158                     puts("checkEND: Different value between
159 START and END packets");
160                     return 1;
161                 }
162                 else {
163                     j++;
164                 }
165             }
166             return 0;
167         }
168         else {
169             puts("checkEND: First END packet byte is not CTEND
170 flag");
171             return 1;
172         }
173     }
174 }
175
176 unsigned char* assembleDataPacket(unsigned char* message,
177 unsigned int messageSize, unsigned int *packetSize){
178
179     *packetSize = messageSize - 4;
180     unsigned char* packet = (unsigned char *)malloc(*packetSize)

```

```
181     return packet;
182 }
183
184 void createFile(unsigned char* data, unsigned int fileSize, char
    *filename){
185     FILE *file = fopen(filename, "wb");
186     fwrite(data, 1, fileSize, file);
187     puts("New file created!");
188     printf("FileSize written: %u\n", fileSize);
189     fclose(file);
190 }
```



## 10.12 macros.h

```
1
2 #define BAUDRATE B38400
3 #define MODEMDEVICE "/dev/ttyS1"
4 #define _POSIX_SOURCE 1 /* POSIX compliant source */
5 #define FALSE 0
6 #define TRUE 1
7
8 #define FLAG 0x7e
9 #define A_EE 0x03 //campo de endereco em comandos enviados pelo
    emissor e respostas enviadas pelo recetor
10 #define A_ER 0x01 //campo de endereco em comandos enviados pelo
    recetor e repostas enviadas pelo emissor
11 #define C_SET 0x03
12 #define C_UA 0x07
13 #define C_DISC 0x0B
14 #define NS0 0x00 //valor de ns na primeira trama do write
15 #define NS1 0x40 //valor de ns segunda trama do write
16 #define ESCAPE_BYTE 0x7d
17 #define ESCAPE_FLAG 0x5e
18 #define ESCAPE_ESCAPE 0x5d
19 #define RR0 0x05
20 #define RR1 0x85
21 #define REJ0 0x01
22 #define REJ1 0x81
23
24 #define TRANSMITTER 0
25 #define RECEIVER 1
26
27 // Macros for Control Packet
28 #define CT_START 0x02
29 #define CT_END 0x03
30 #define T1 0x00
31 #define T2 0x01
32
33 // Macros for Data Packet
34 #define CONTROL 0x01
35 #define PACKETSIZE 128
36
37 #define ERROR -1
38 #define MAXTRIES 10
39 #define TIMEOUT 1 //Time to wait for a response from the
    receiver
40
41 #define BCC1ERRORRATE 0
42 #define BCC2ERRORRATE 0
```

### 10.13 Anexo I - Principais funções utilizadas no Protocolo de ligação Lógica(documentação encontra-se em llfunctions.h)

```
1 int llopen(int fd, int status);  
2  
3 int llwrite(int fd, unsigned char *buffer, int length);  
4  
5 unsigned int llread(int fd, unsigned char *buffer);  
6  
7 int llclose(int fd, int status);
```

## 10.14 Anexo I - Principais funções utilizadas na camada de Aplicação(documentação encontra-se em applications.h)

```
1 unsigned char* openFile(char *filename, off_t *fileSize);
2
3 unsigned char* parseControlPacket(unsigned int state, off_t
   fileSize, char* filename, int sizeFileName, int *
   sizeControlPacket);
4
5 unsigned char* parseDataPacket(unsigned char *message, off_t
   fileSize, int *length);
6
7 unsigned char* splitPacket(unsigned char *message, off_t *index,
   int *packetSize, off_t fileSize);
8
9 int checkStart(unsigned char* start, unsigned int *filesize,
   char *name, unsigned int *nameSize);
10
11 unsigned char* assembleDataPacket(unsigned char* message,
   unsigned int messageSize, unsigned int *packetSize);
```

### 10.15 Variação do Baudrate

C	Time	R	S
4800	130,604498	2220,419698	0,462587437
9600	65,486258	4428,361138	0,461287619
19200	32,9410888	8803,497716	0,458515506
38400	16,646928	17420,43937	0,453657275
57600	11,264985	25743,20339	0,446930614

Figure 1: Tabela com diferentes valores de Baudrate

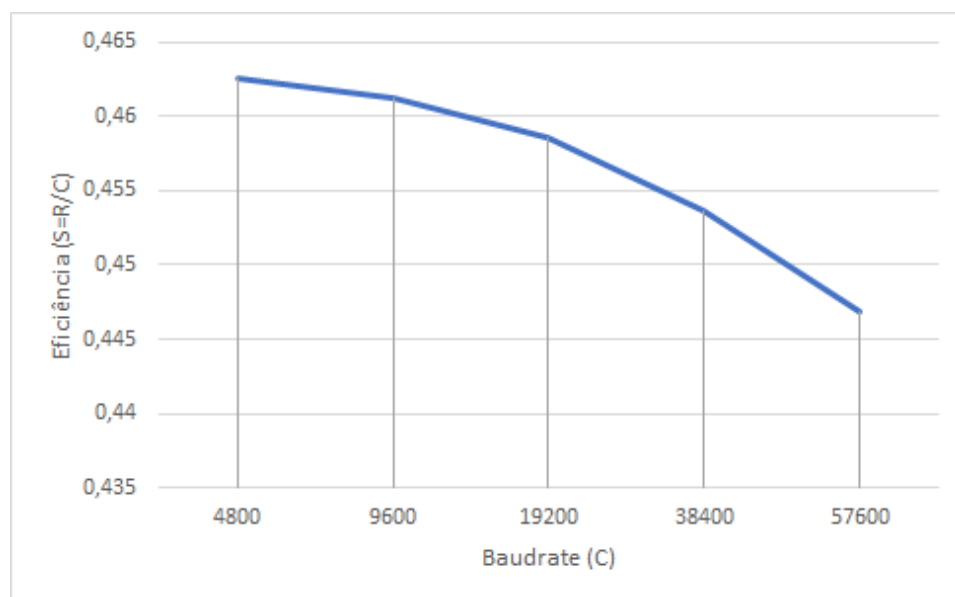


Figure 2: Eficiência em função do Baudrate

### 10.16 Variação do tamanho das tramas

Tamanho	Time	R	S
32	16,646928	17420,43937	0,453657275
64	13,153735	22046,72665	0,574133507
96	12,014862	24136,50694	0,628554868
128	11,401286	25435,44649	0,662381419
160	11,082233	26167,7227	0,681451112
192	10,823469	26793,33216	0,697743025
224	10,660243	27203,58251	0,708426628

Figure 3: Tabela com diferentes tamanhos da trama de envio

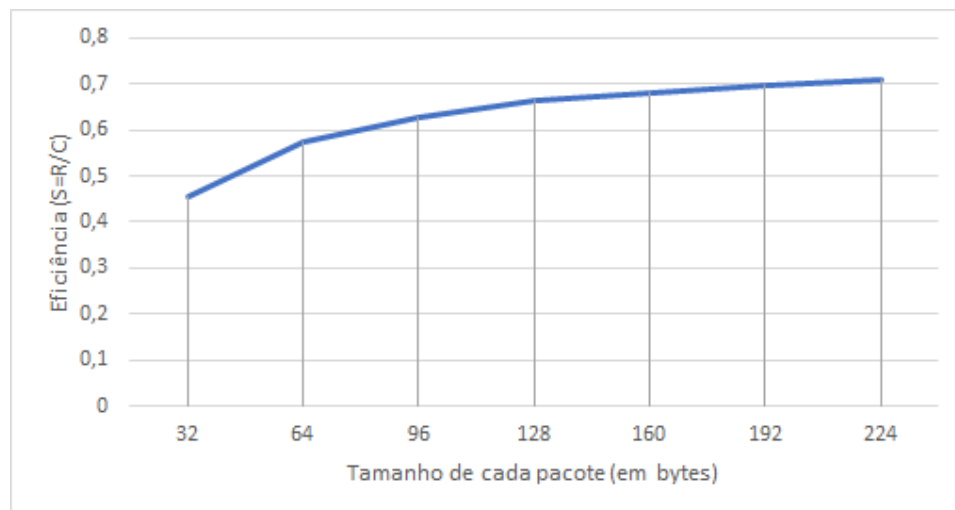


Figure 4: Eficiência em função do tamanho da trama de envio

### 10.17 Introdução de atraso de propagação

Atraso	Time	R	S
0,001	10,799957	26851,66246	0,699262043
0,05	14,439983	20082,90453	0,522992305
0,1	28,690873	10107,63249	0,263219596
0,15	42,946874	6752,454207	0,175845162

Figure 5: Tabela com diferentes valores de atraso no envio de tramas

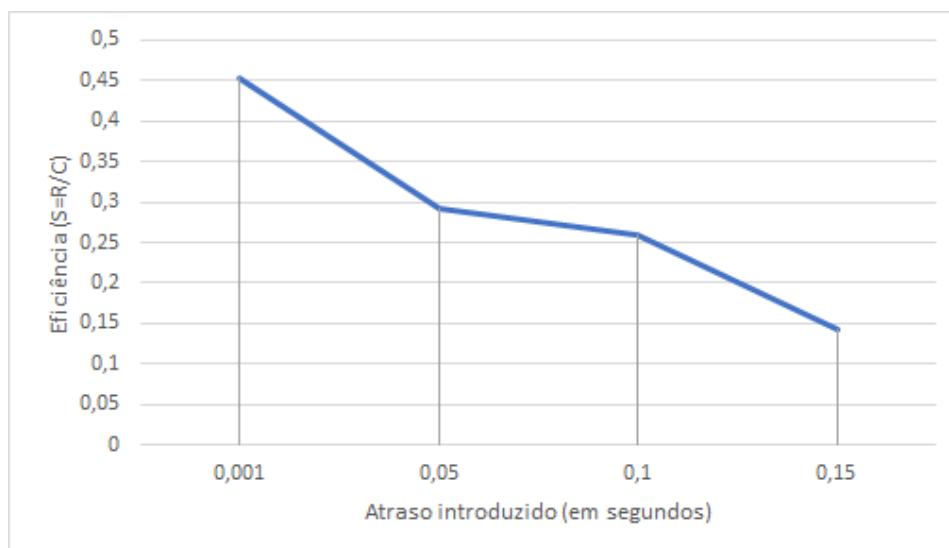


Figure 6: Eficiência em função do atraso introduzido

### 10.18 Introdução de erros nas tramas

Erro(%)	Time	R	S
0+0	16,646928	17420,43937	0,453657275
1+1	25,969783	11166,70093	0,290799503
3+3	29,259915	9911,060917	0,258100545
5+5	52,640816	5508,972353	0,143462822
7+7	87,051704	3331,316754	0,08675304

Figure 7: Tabela com diferentes valores percentuais de erro introduzido

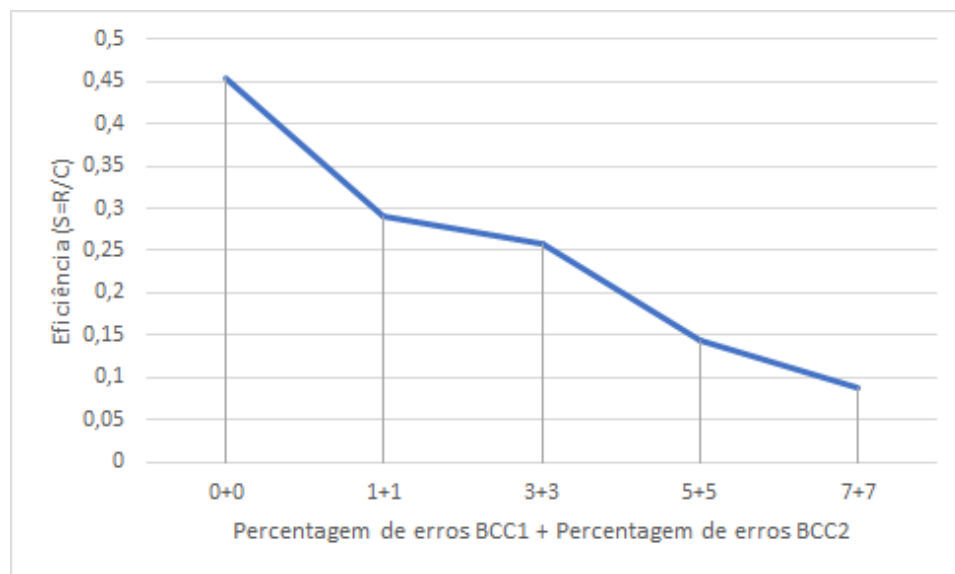


Figure 8: Eficiência em função da percentagem de erros introduzidos no BCC e BCC2