

Protocolo de Ligação de Dados

Primeiro Trabalho Laboratorial

Hugo Miguel Monteiro Guimarães
Pedro Varandas da Costa Azevedo da Ponte

Trabalho realizado no âmbito da
Unidade Curricular de Redes de Computadores



Mestrado Integrado em Engenharia Informática e Computação
Faculdade de Engenharia da Universidade do Porto

Porto

17 de novembro de 2020

Contents

1	Introdução	3
2	Arquitetura	4
3	Estrutura do Código	4
4	Casos De Uso Principais	6
5	Protocolo De Ligação Lógica	6
5.1	LLOPEN	6
5.2	LLWRITE	7
5.3	LLREAD	7
5.4	LLCLOSE	8
6	Protocolo De Aplicação	8
7	Validação	9
8	Eficiência Do Protocolo De Ligação De Dados	9
8.1	Variação da capacidade de Baudrate	9
8.2	Variação do tamanho das tramas	9
8.3	Atraso no envio das tramas	10
8.4	Geração de erros no cabeçalho e no campo de dados	10
9	Conclusões	10
10	Anexos	12
10.1	Anexo I - emissor.h	12
10.2	Anexo II - emissor.c	13
10.3	Anexo III - recetor.h	16
10.4	Anexo IV - recetor.c	17
10.5	Anexo V - llfunctions.h	20
10.6	Anexo VI - llfunctions.c	22
10.7	Anexo VII - stateMachines.h	31
10.8	Anexo VIII - stateMachines.c	32
10.9	Anexo IX - application.h	43
10.10	Anexo X - application.c	46
10.11	Anexo XI - macros.h	51

10.12 Anexo XII - Principais funções utilizadas no Protocolo de ligação Lógica	52
10.13 Anexo XIII - Principais funções utilizadas na camada de Aplicação	53
10.14 Anexo XIV - Variação do Baudrate	54
10.15 Anexo XV - Variação do tamanho das tramas	55
10.16 Anexo XVI - Introdução de atraso de propagação	56
10.17 Anexo XVII - Introdução de erros nas tramas	57

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.

stateMachines.c

- **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.

application.c

- **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.

Para mais informações, consultar os anexos no final do relatório (Anexos I a XI).

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, seleciona 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 selecionar 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 o anexo XII para mais informações sobre as principais funções utilizadas para implementação do Protocolo de Ligação Lógica.

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*.

Consultar anexo V para mais informações sobre a implementação desta função.

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.

Consultar anexo V para mais informações sobre a implementação desta função.

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.

Consultar anexo V para mais informações sobre a implementação desta função.

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.

Consultar anexo V para mais informações sobre a implementação desta função.

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 o anexo XIII para mais informações sobre as principais funções utilizadas para implementação do Protocolo de Aplicação.

Para que tal fosse implementado, recorreremos 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 relacionados com a 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.

No anexo XIV encontram-se os dados por nós recolhidos e o respetivo gráfico que nos permitiram chegar a estas conclusões.

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.

No anexo XV encontram-se os dados por nós recolhidos e o respetivo gráfico que nos permitiram chegar a estas conclusões.

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.

No anexo XVI encontram-se os dados por nós recolhidos e o respetivo gráfico que nos permitiram chegar a estas conclusões.

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.

No anexo XVII encontram-se os dados por nós recolhidos e o respetivo gráfico que nos permitiram chegar a estas conclusões.

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 Anexo I - 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);
```

10.2 Anexo II - 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     // Cicle to send 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 }
```


10.3 Anexo III - 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"
```

10.4 Anexo IV - 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 }

```

10.5 Anexo V - 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 sucess, -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 *mensagem, 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
        for the termination protocl
62 * @return returns 0 upon sucess, -1 otherwise
63 */
64 int llclose(int fd, int status);
65
66 /**
67 * \brief handles the alarm
68 * @param signo signal number to be handled
69 */
70 void alarmHandler(int signo);
71
72
73 unsigned char* generateRandomBCC(unsigned char* packet, int
        packetSize);
74
75 unsigned char* generateRandomBCC2(unsigned char* packet, int
        packetSize);

```

10.6 Anexo VI - 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         else if(status == RECEIVER) {
78             if(readSetMessage(fd) == TRUE) {
79                 puts("RECEIVER: Read SET message correctly");
80                 if(sendMessage(fd, C_UA) == -1) {
81                     fprintf(stderr, "llopen - Error writing to
82 serial port (Receiver)\n");
83                     return ERROR;
84                 }
85             }
86             else {
87                 puts("RECEIVER: Sent UA message");
88             }
89
90         }
91         else {
92             fprintf(stderr, "llopen - Error reading from serial
93 port (Receiver)\n");
94             return ERROR;
95         }
96     }

```



```

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     unsigned char bcc2;
129     unsigned int sizebcc2 = 1;
130     unsigned int messageSize = length+6;
131     unsigned char *bcc2Stuffed = (unsigned char *)malloc(sizeof(
132     unsigned char));
133     unsigned char *message = (unsigned char *)malloc(messageSize
134     * sizeof(unsigned char));
135
136     bcc2 = getBCC2(buffer, length);

```

```

135     bcc2Stuffed = stuffBCC2(bcc2, &sizebcc2);
136
137
138     // Start to fill the message
139     message[0] = FLAG;
140     message[1] = A_EE;
141     if(trama == 0){
142         message[2] = NS0;
143     }
144     else{
145         message[2] = NS1;
146     }
147     message[3] = message[1] ^ message[2];
148
149     // Start to read from 4 and size depends on the message to
send
150     int i = 4;
151     for(int j = 0; j < length; j++){
152         if(buffer[j] == FLAG){
153             message = (unsigned char *)realloc(message, ++
messageSize);
154             message[i] = ESCAPE_BYTE;
155             message[i + 1] = ESCAPE_FLAG;
156             i+=2;
157         }
158         else if(buffer[j] == ESCAPE_BYTE){
159             message = (unsigned char *)realloc(message, ++
messageSize);
160             message[i] = ESCAPE_BYTE;
161             message[i+1] = ESCAPE_ESCAPE;
162             i+=2;
163         }
164         else{
165             message[i] = buffer[j];
166             i++;
167         }
168     }
169
170     if(sizebcc2 == 2){
171         message = (unsigned char *)realloc(message, ++
messageSize);
172         message[i] = bcc2Stuffed[0];
173         message[i + 1] = bcc2Stuffed[1];
174         i+=2;
175     }
176     else{
177         message[i] = bcc2;
178         i++;
179     }

```

```

180     message[i] = FLAG;
181
182     // Packet I filled
183     // printMessage
184     for(int j = 0; j < messageSize; j++){
185         printf("message[%d] = 0x%X\n", j, message[j]);
186     }
187
188
189     counter = 0;
190     STP = FALSE;
191
192     // Send packet
193     do {
194
195         unsigned char* copyBcc = (unsigned char *)malloc(
messageSize);
196         unsigned char* copyBcc2 = (unsigned char *)malloc(
messageSize);
197
198         copyBcc = generateRandomBCC(message, messageSize);
199         copyBcc2 = generateRandomBCC2(copyBcc, messageSize);
200
201
202         int wr = write(fd, copyBcc2, messageSize);
203
204         printf("TRANSMITTER: SET message sent: %d bytes sent\n",
wr);
205
206         alarm(TIMEOUT);
207
208         readReceiverMessage(fd);
209
210         // Handle rcv
211         if((rcv == RR0 && trama == 1) || (rcv == RR1 && trama ==
0)) {
212             counter = 0;
213             trama = (trama + 1) % 2;
214             STP = FALSE;
215             alarm(0);
216             if(rcv == RR0) {
217                 puts("TRANSMITTER: Received RR0");
218             }
219             else {
220                 puts("TRANSMITTER: Received RR1");
221             }
222             break;
223         }
224

```

```

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

```

```

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

```

```

318     message (Receiver)\n");
319         return ERROR;
320     }
321     tcsetattr(fd, TCSANOW, &oldtio);
322 }
323 return 0;
324 }
325
326
327 void alarmHandler(int signo){
328
329     counter++;
330     if(counter >= MAXTRIES){
331         printf("Exceeded maximum amount of tries: (%d)\n", MAXTRIES)
332         ;
333         exit(0);
334     }
335     return ;
336 }
337
338
339 unsigned char* generateRandomBCC(unsigned char* packet, int
packetSize){
340     unsigned char* copy = (unsigned char *)malloc(packetSize);
341     memcpy(copy, packet, packetSize);
342
343     if(((rand() % 100) + 1) <= BCCIERRORRATE){
344         unsigned char hex[16] = {'0', '1', '2', '3', '4', '5', '6', '7',
'8', '9', 'A', 'B', 'C', 'D', 'E', 'F'};
345
346         copy[(rand() % 3) + 1] = hex[rand() % 16];
347         puts("BCC Value sucessfully changed");
348     }
349     return copy;
350 }
351
352
353 unsigned char* generateRandomBCC2(unsigned char* packet, int
packetSize){
354     unsigned char* copy = (unsigned char *)malloc(packetSize);
355     memcpy(copy, packet, packetSize);
356
357     if(((rand() % 100) + 1) <= BCC2ERRORRATE){
358         unsigned char hex[16] = {'0', '1', '2', '3', '4', '5', '6', '7',
'8', '9', 'A', 'B', 'C', 'D', 'E', 'F'};
359
360         copy[(rand() % (packetSize - 5)) + 4] = hex[rand() %

```

```
16];  
361     puts("BCC2 Value sucessfully changed");  
362 }  
363 return copy;  
364 }
```

10.7 Anexo VII - 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 ,
    unsigned int *size);
30
31 int receiveDISC(int fd);
32
33 int checkBCC2(unsigned char *packet, int size);
```


10.8 Anexo VIII - 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 return TRUE;
230
231 }
232
233
234 int receiverRead_StateMachine(int fd, unsigned char* frame,
235                               unsigned int *size) {
236
237     unsigned char buf, check = 0;
238     int trama = 0;
239     enum state current = START;
240     int correctBCC2 = FALSE; // if no errors in BCC2,
241                             // correctBCC2 = 1; else correctBCC2 = 0
242     int errorOnDestuffing = FALSE; // if no errors occur on

```

```

241 destuffing, the var stays equal to 0, else the value is 1
242 counter_errors++;
243 puts("Receiver reading frames");
244 while(current != STOP) {
245
246     res = read(fd, &buf, 1);
247     printf("read : 0x%X\n", buf);
248
249     if(res == ERROR) {
250         fprintf(stderr, "llread() - Error reading from
buffer");
251         return -1;
252     }
253
254     switch (current)
255     {
256     case START:
257         if(buf == FLAG) {
258             current = FLAG_RCV;
259         }
260         break;
261
262     case FLAG_RCV:
263         if(buf == A_EE) {
264             current = A_RCV;
265         }
266         else if(buf != FLAG) {
267             current = START;
268         }
269         break;
270
271     case A_RCV:
272         if(buf == NS0) {
273             current = C_RCV;
274             check = buf;
275             trama = 0;
276         }
277
278         else if(buf == NS1) {
279             current = C_RCV;
280             check = buf;
281             trama = 1;
282         }
283
284         else if(buf == FLAG) {
285             current = FLAG_RCV;
286         }
287

```

```

288         else {
289             current = START;
290         }
291         break;
292
293     case C_RCV:
294         if(buf == (A_EE ^ check)) {
295             current = BCC_OK;
296         }
297
298         else if(buf == FLAG) {
299             current = FLAG_RCV;
300         }
301
302         else {
303             current = START;
304         }
305         break;
306
307     case BCC_OK:
308         if(buf == FLAG) {
309             if(checkBCC2(frame, *size) != ERROR) {
310                 correctBCC2 = TRUE;
311                 current = STOP;
312             }
313             else {
314                 correctBCC2 = FALSE;
315                 current = STOP;
316             }
317         }
318         else if(buf == ESCAPE_BYTE) {
319             current = BYTE_DESTUFFING;
320         }
321         else {
322             (*size)++;
323             frame = (unsigned char *)realloc(frame, *size);
324             frame[*size - 1] = buf; // still receiving data
325         }
326     }
327
328     break;
329
330
331     case BYTE_DESTUFFING:
332         if(buf == ESCAPE_FLAG) {
333             frame = (unsigned char *)realloc(frame, ++(*size
334 ));
335             frame[*size - 1] = FLAG;
336         }

```

```

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

```



```

383         //send RR(Nr = 0)
384         sendMessage(fd, RR0);
385         puts("Receiver send RR0 after repeated
information");
386     }
387     else {
388         //send RR(Nr = 1)
389         sendMessage(fd, RR1);
390         puts("Receiver send RR1 after repeated
information");
391     }
392 }
393 }
394 else { //error in BCC2 or interferences
395     if(trama != expectedTrama) {
396         if(trama == 0) {
397             //send RR(Nr = 1)
398             sendMessage(fd, RR1);
399             expectedTrama = 1;
400             puts("Receiver send RR1 after errors in BCC2");
401         }
402         else {
403             //send RR(Nr=0)
404             sendMessage(fd, RR0);
405             expectedTrama = 0;
406             puts("Receiver send RR0 after errors in BCC2");
407         }
408     }
409
410     else { //correct trama, but with error in BCC2
411         *size = 0;
412
413         if(trama == 0) {
414             //send REJ 0
415             sendMessage(fd, REJ0);
416             expectedTrama = 0;
417             puts("Receiver send REJ0");
418         }
419
420         else {
421             //send REJ1
422             sendMessage(fd, REJ1);
423             expectedTrama = 1;
424             puts("Receiver send REJ1");
425         }
426     }
427 }
428 return 0;
429 }

```

```

430
431 int receiveDISC(int fd) {
432
433     enum state current = START;
434
435     int finish = FALSE;
436     unsigned char r;
437
438     while (finish == FALSE)
439     {
440         res = read(fd, &r, 1);
441
442         if(res == 0) {
443             finish = TRUE;
444         }
445
446         switch (current)
447         {
448             case START:
449                 if (r == FLAG){
450
451                     current = FLAG.RCV;
452                 }
453                 break;
454             case FLAG.RCV:
455                 if (r == A.EE){
456                     current = A.RCV;
457                 }
458                 else if (r == FLAG){
459                     current = FLAG.RCV;
460                 }
461                 else{
462                     current = START;
463                 }
464                 break;
465             case A.RCV:
466                 if (r == C.DISC){
467                     current = C.RCV;
468                 }
469                 else if (r == FLAG){
470                     current = FLAG.RCV;
471                 }
472                 else{
473                     current = START;
474                 }
475                 break;
476             case C.RCV:
477                 if (r == (C.DISC ^ A.EE)){
478                     current = BCC.OK;

```

```

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

```

10.9 Anexo IX - 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);

```

10.10 Anexo X - 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 unsigned char* parseDataPacket(unsigned char *message, off_t
72     fileSize , int *length){
73
74     unsigned char *packet = (unsigned char*)malloc( fileSize + 4)
75     ;
76
77     packet[0] = CONTROL;
78     packet[1] = packetNumber % 255;
79     packet[2] = fileSize / 256;
80     packet[3] = fileSize % 256;
81
82     // Fill packet using message
83     for(int i = 0; i < *length; i++){
84         packet[4 + i] = message[i];
85     }
86
87     *length += 4;
88     packetNumber++;
89
90     return packet;
91 }

```



```

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 }
```

10.11 Anexo XI - 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 //commands sent by emissor and answers sent by
    recetor
10 #define A_ER 0x01 //commands sent by recetor and answers sent by
    emissor
11 #define C_SET 0x03
12 #define C_UA 0x07
13 #define C_DISC 0x0B
14 #define NS0 0x00
15 #define NS1 0x40
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 repsonse from the receiver
40
41 #define BCCIERRORRATE 0
42 #define BCC2ERRORRATE 0
```

10.12 Anexo XII - Principais funções utilizadas no Protocolo de ligação Lógica

```
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);
```

10.13 Anexo XIII - Principais funções utilizadas na camada de Aplicação

```
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);
```

10.14 Anexo XIV - 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

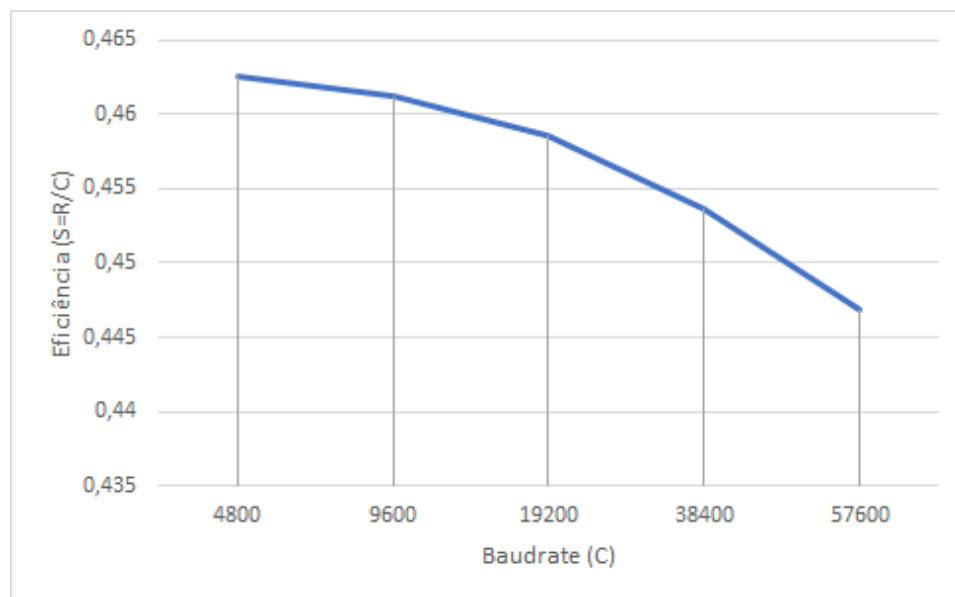


Figure 2: Eficiência em função do Baudrate

10.15 Anexo XV - 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

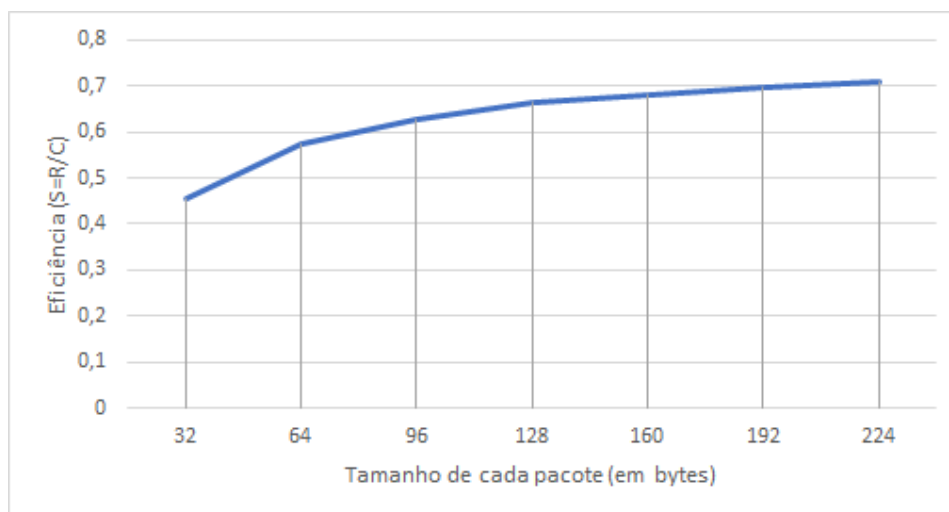


Figure 4: Eficiência em função do tamanho da trama de envio

10.16 Anexo XVI - 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

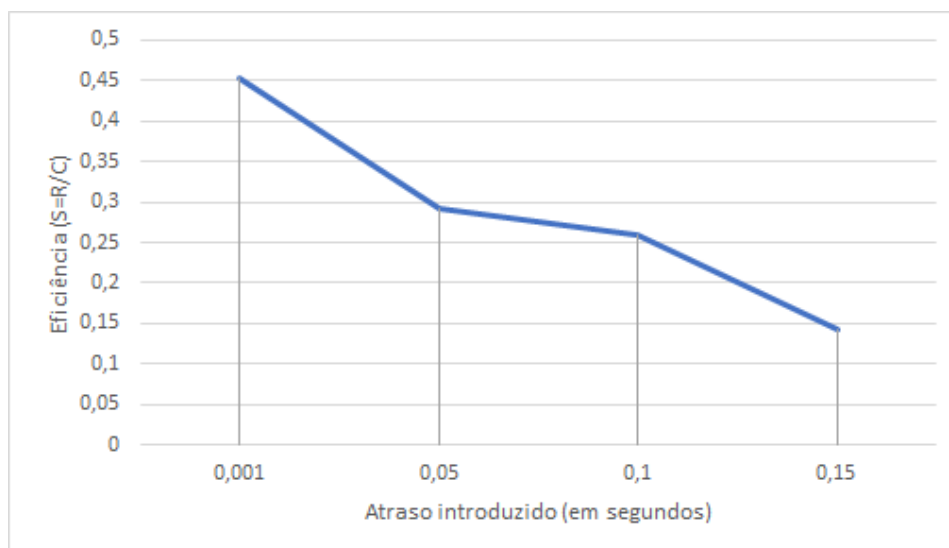


Figure 6: Eficiência em função do atraso introduzido

10.17 Anexo XVII - 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

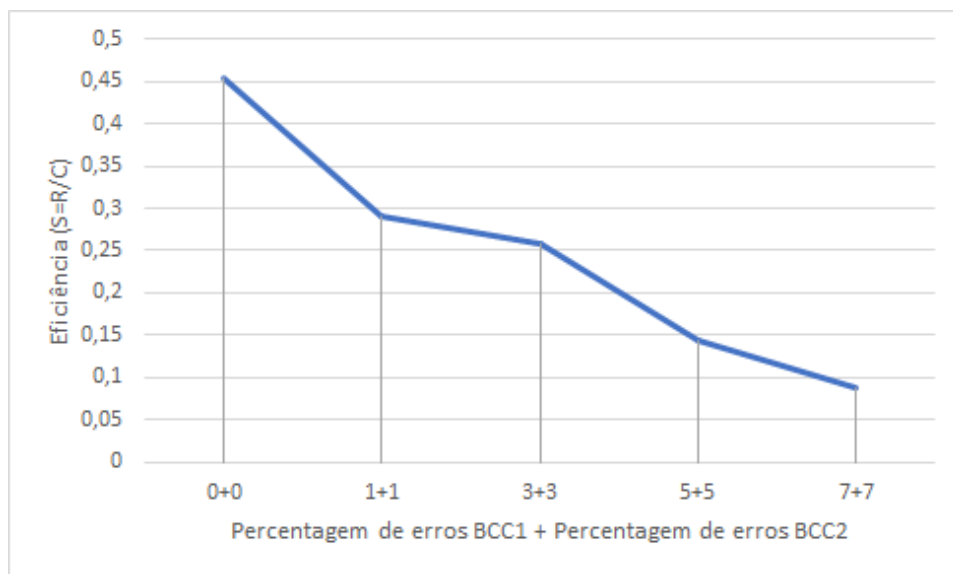


Figure 8: Eficiência em função da percentagem de erros introduzidos no BCC e BCC2