

Primeiro Trabalho Laboratorial

Hugo Guimarães, Pedro Ponte

November 16, 2020

1 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.

2 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:

2.1 Arquitetura

Descrição dos blocos funcionais e interfaces implementados.

2.2 Estrutura do Código

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

2.3 Casos De Uso Principais

Identificação dos principais casos de uso e da sequência de chamada de funções.

2.4 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.

2.5 Protocolo De Aplicação

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

2.6 Validação

Descrição dos testes efetuados com apresentação quantificada dos resultados.

2.7 Eficiência Do Protocolo De Ligação De Dados

Caracterização estatística da eficiência do protocolo de Stop&Wait implementado.

2.8 Conclusões

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

3 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.

4 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*.

Variáveis globais

- **STP**
- **counter** - Contador de chamadas ao alarmHandler, inicializada a 0.
- **trama** - Representa o número sequencial da trama enviada pelo emissor (*Ns*), inicializada a 0.

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.

Variáveis globais

- **STP**
- **counter** - Contador de chamadas ao alarmHandler, inicializada a 0.

- **trama** - Representa o número sequencial da trama enviada pelo emissor (Ns), inicializada a 0.

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. **Variáveis globais**
 - **packetNumber** - Contagem do número de pacotes enviados.

5 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 2 dispositivos ocorre da seguinte forma:

- 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.

6 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`.

6.1 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*.

6.2 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.

6.3 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.

6.4 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.

7 Protocolo De Aplicação

O protocolo de aplicação contém as seguintes funcionalidades:

- Leitura da informação sobre o ficheiro a enviar;
- Geração e leitura de pacotes de controlo do tipo *START* e *END*, contendo o tamanho e o nome do ficheiro;
- Divisão do ficheiro em fragmentos mais pequenos;
- Preenchimento de um fragmento do ficheiro com um cabeçalho de controlo;
- Leitura do ficheiro a criar, do lado do recetor, e criação do mesmo sem alterações, por parte do recetor;

Para que tal fosse implementado, recorreremos às funções descritas a seguir.

7.1 OpenFile

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

7.2 ParseControlPacket

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

7.3 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.

7.4 SplitPacket

Divide o ficheiro em fragmentos mais pequenos.

7.5 CheckStart

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

7.6 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.

7.7 AssembleDataPacket

Retorna o campo de dados de um pacote.

7.8 CreateFile

Gera um ficheiro de acordo com os dados recebidos.

8 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, recorreremos 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.

9 Eficiência Do Protocolo De Ligação De Dados

9.1 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.

9.2 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.

9.3 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.

9.4 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.

10 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.

11 Anexos

11.1 Código

11.2 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);
```

11.3 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) && (strcmp(
11         "/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;
```

```

21 clock_gettime(CLOCK_REALTIME, &initialTime);
22
23 if ((fd = open(argv[1], ORDWR | O_NOCTTY)) < 0) {
24     perror(argv[1]);
25     return -2;
26 }
27
28
29 int fileNameSize = strlen(argv[2]);
30 char* filename = (char*)malloc(fileNameSize);
31 filename = (char*)argv[2];
32 off_t fileSize = 0;
33 int sizeControlPacket = 0;
34
35 unsigned char *data = openFile(filename, &fileSize);
36
37 // Dealing with the SET and UA
38 if(llopen(fd, TRANSMITTER) == ERROR){
39     puts("TRANSMITTER: Error on llopen");
40     return -3;
41 }
42
43 // Start Control packet
44 unsigned char *start = parseControlPacket(CT_START, fileSize,
45     filename, fileNameSize, &sizeControlPacket);
46
47 if(llwrite(fd, start, sizeControlPacket) != TRUE){
48     puts("TRANSMITTER: Error writing START control packet");
49     return -4;
50 }
51 free(start);
52
53 // Ciclo de envio dos packets
54 int packetSize = PACKETSZ;
55 off_t index = 0;
56
57 while(index < fileSize && packetSize == PACKETSZ){
58     unsigned char* packet = splitPacket(data, &index, &packetSize,
59         fileSize);
60
61     int length = packetSize;
62
63     unsigned char* message = parseDataPacket(packet, fileSize, &
64         length);
65
66     if(llwrite(fd, message, length) != TRUE){
67         puts("TRANSMITTER: Error sending data packet");
68         return -5;
69     }
70
71     printf("Sent packet number: %d\n", packetNumber);
72
73     free(message);
74 }
75
76 // End Control packet

```

```

75 unsigned char *end = parseControlPacket(CT_END, fileSize,
    filename, fileNameSize, &sizeControlPacket);
76
77 if(!llwrite(fd, end, sizeControlPacket) != TRUE ){
78     puts("TRANSMITTER: Error writing END control packet");
79     return -6;
80 }
81 free(end);
82
83
84 if(!llclose(fd, TRANSMITTER) == ERROR){
85     puts("TRANSMITTER: Error on llclose");
86     return -7;
87 }
88
89 clock_gettime(CLOCK_REALTIME, &finalTime);
90
91 double accum = (finalTime.tv_sec - initialTime.tv_sec) + (
    finalTime.tv_nsec - initialTime.tv_nsec) / 1E9;
92
93 printf("Seconds passed: %f\n", accum);
94
95 sleep(1);
96 close(fd);
97 free(data);
98
99 return 0;
100 }

```

11.4 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"

```

11.5 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              );
15        return -1;
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
90     for(int i= 0; i < packetSize; i++){
91         result[index + i] = dataPacket[i];
92     }
93
94     index += packetSize;
95
96     free(dataPacket);
97 }
98
99 if(checkEND(start, sizeStart, final, messageSize) == 1) {
100     puts("Start and End packets are different!");
101     return -6;
102 }
103
104 printf("Received a file of size: %u\n", fileSize);
105
106 // Creating the file to be rewritten after protocol
107 createFile(result,fileSize,fileName);
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) + (
    finalTime.tv_nsec - initialTime.tv_nsec) / 1E9;
117 printf("Seconds passed: %f\n", accum);
118 sleep(1);
119
120 free(fileName);
121 free(result);
122
123 close(fd);
124
125 return 0;
126 }

```

11.6 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 the
19  *         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 *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  */

```

```

39 unsigned char* stuffBCC2(unsigned char bcc2, unsigned int *size);
40
41 /**
42  * \brief Sends an I packet from a message from buffer to the
43  *       serial port
44  * @param fd file descriptor of the serial port
45  * @param buffer containing the message to be sent
46  * @param length length of the message to be sent
47  * @return TRUE(1) upon success, FALSE(0) upon failure
48  */
49 int llwrite(int fd, unsigned char *buffer, int length);
50
51 /**
52  * \brief Reads an I packets sent through the serial port
53  * @param fd file descriptor for the serial port
54  * @param buffer buffer read from the serial port
55  * @return size of the read buffer
56  */
57 unsigned int llread(int fd, unsigned char *buffer);
58
59 /**
60  * \brief Termination of the protocol by serial port
61  * @param fd file descriptor of the serial port
62  * @param status if 0, acts as sender. if 1, acts as receiver for
63  *       the termination protocol
64  * @return returns 0 upon success, -1 otherwise
65  */
66 int llclose(int fd, int status);
67
68 /**
69  * \brief handles the alarm
70  * @param signo signal number to be handled
71  */
72 void alarmHandler(int signo);
73
74 unsigned char* generateRandomBCC(unsigned char* packet, int
75                                 packetSize);
76
77 unsigned char* generateRandomBCC2(unsigned char* packet, int
78                                   packetSize);

```

11.7 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 unused
30     */
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
49     if(status == TRANSMITTER) {
50
51         // Installing Alarm Handler
52         if(signal(SIGALRM, alarmHandler) || siginterrupt(SIGALRM,
53         1)){
54             puts("Signal instalation failed");
55             return ERROR;
56         }
57
58         counter = 0;
59         do{
60             int wr;
61             if((wr = sendMessage(fd, C_SET)) != ERROR){
62                 printf("llopen: C_SET message sent: %d \n", wr);
63             }
64             else{
65                 puts("llopen: Error sending message");
66             }
67
68             alarm(TIMEOUT); // Call an alarm to wait for the

```

```

65     message
66         if (receiveUA (fd) == TRUE){
67             puts("TRANSMITTER: UA received\n");
68             STP = TRUE;
69             counter = 0;
70             alarm(0);
71         }
72     } while (STP == FALSE && counter < MAXTRIES);
73 }
74 else if (status == RECEIVER) {
75     if (readSetMessage (fd) == TRUE) {
76         puts("RECEIVER: Read SET message correctly");
77         if (sendMessage (fd, CUA) == -1) {
78             fprintf(stderr, "llopen - Error writing to serial
79 port (Receiver)\n");
80             return ERROR;
81         }
82         else {
83             puts("RECEIVER: Sent UA message");
84         }
85     }
86     else {
87         fprintf(stderr, "llopen - Error reading from serial
88 port (Receiver)\n");
89         return ERROR;
90     }
91 }
92 return 0;
93 }
94 unsigned char getBCC2(unsigned char *mensagem, int size){
95     unsigned char bcc2 = mensagem[0];
96     for(int i = 1; i < size; i++){
97         bcc2 ^= mensagem[i];
98     }
99     return bcc2;
100 }
101 }
102 }
103 unsigned char* stuffBCC2(unsigned char bcc2, unsigned int *size){
104     unsigned char* stuffed;
105     if (bcc2 == FLAG){
106         stuffed = malloc(2 * sizeof(unsigned char));
107         stuffed[0] = ESCAPE_BYTE;
108         stuffed[1] = ESCAPE_FLAG;
109         (*size) = 2;
110     }
111     else if (bcc2 == ESCAPE_BYTE){
112         stuffed = malloc(2 * sizeof(unsigned char));
113         stuffed[0] = ESCAPE_BYTE;
114         stuffed[1] = ESCAPE_ESCAPE;
115         (*size) = 2;
116     }
117     else{
118

```

```

119     stuffed = malloc(sizeof(unsigned char));
120     stuffed[0] = bcc2;
121     (*size) = 1;
122 }
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
129     unsigned char bcc2;
130     unsigned int sizebcc2 = 1;
131     unsigned int messageSize = length+6;
132     unsigned char *bcc2Stuffed = (unsigned char *)malloc(sizeof(
133     unsigned char));
134     unsigned char *message = (unsigned char *)malloc(messageSize *
135     sizeof(unsigned char));
136
137     bcc2 = getBCC2(buffer, length);
138     bcc2Stuffed = stuffBCC2(bcc2, &sizebcc2);
139
140     // Inicio do preenchimento da mensagem
141     message[0] = FLAG;
142     message[1] = A.EE;
143     if(trama == 0){
144         message[2] = NS0;
145     }
146     else{
147         message[2] = NS1;
148     }
149     message[3] = message[1] ^ message[2];
150
151     // Comeca a ler do 4 e o tamanho depende da mensagem a ser
152     // enviada
153     int i = 4;
154     for(int j = 0; j < length; j++){
155         if(buffer[j] == FLAG){
156             message = (unsigned char *)realloc(message, ++
157             messageSize);
158             message[i] = ESCAPE.BYTE;
159             message[i + 1] = ESCAPE.FLAG;
160             i+=2;
161         }
162         else if(buffer[j] == ESCAPE.BYTE){
163             message = (unsigned char *)realloc(message, ++
164             messageSize);
165             message[i] = ESCAPE.BYTE;
166             message[i+1] = ESCAPE.ESCAPE;
167             i+=2;
168         }
169         else{
170             message[i] = buffer[j];
171             i++;
172         }
173     }

```

```

170
171     if (sizebcc2 == 2){
172         message = (unsigned char *)realloc(message, ++messageSize);
173         message[i] = bcc2Stuffed[0];
174         message[i + 1] = bcc2Stuffed[1];
175         i+=2;
176     }
177     else{
178         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 RR1");
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 ");
243         STP = TRUE;
244     }
245
246     else {
247         puts("TRANSMITTER: Received an invalid message");
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         alarm(TIMEOUT); // Call an alarm to wait for the
message
287
288         if(receiveDISC(fd) == 0 && res != 0){
289             puts("TRANSMITTER: C_DISC received");
290             STP = TRUE;
291             counter = 0;
292             alarm(0);
293         }
294         while(STP == FALSE && counter < MAXTRIES);
295
296         if(sendMessage(fd, C_UA)) {
297             puts("TRANSMITTER: Send UA");
298         }
299         tcsetattr(fd, TCSANOW, &oldtio);
300     }
301
302 //recetor:
303 // le a mensagem DISC enviada pelo emissor, envia DISC e recebe UA
304     else if(status == RECEIVER) {
305         if (receiveDISC(fd) == 0) {
306             puts("RECEIVER: Read DISC");
307             if(sendMessage(fd, C_DISC)) {
308                 puts("RECEIVER: Send DISC");
309                 if(receiveUA(fd) == TRUE) {
310                     puts("RECEIVER: Read UA");
311                 }
312             }
313             else {
314                 fprintf(stderr, "llclose- Error reading UA
message (Receiver)\n");
315                 return ERROR;
316             }
317         }
318         else {
319             fprintf(stderr, "llclose- Error writing DISC
message to serial port (Receiver)\n");
320             return ERROR;
321         }
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) <= BCC1ERRORRATE){
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 }

```

11.8 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,
    unsigned int *size);
30
31 int receiveDISC(int fd);
32
33 int checkBCC2(unsigned char *packet, int size);

```

11.9 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                current = C_RCV;
120                check = r;
121                rcv = r;
122            }
123            else if (r == FLAG){
124                current = FLAG_RCV;
125            }
126            else{
127                current = START;
128            }
129            break;
130        case C_RCV:
131            if (r == (check ^ A_EE)){
132                current = BCC_OK;
133            }
134            else if (r == FLAG){
135                current = FLAG_RCV;
136            }
137            else{
138                current = START;
139            }
140            break;

```

```

141         case BCC_OK:
142             if (r == FLAG){
143                 finish = TRUE;
144             }
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, correctBCC2 =
242                             // 1; else correctBCC2 = 0
243     int errorOnDestuffing = FALSE; // if no errors occur on
244                                     // destuffing, the var stays equal to 0, else the value is 1
245     counter_errors++;
246
247     puts("Receiver reading frames");
248     while(current != STOP) {
249
250         res = read(fd, &buf, 1);
251         printf("read : 0x%X\n", buf);
252
253         if(res == ERROR) {
254             fprintf(stderr, "llread() - Error reading from buffer")

```

```

252         ;
253         return -1;
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     }
320     else if(buf == ESCAPE_BYTE) {
321         current = BYTE_DESTUFFING;
322     }
323     else {
324         (*size)++;
325         frame = (unsigned char *)realloc(frame, *size);
326         frame[*size - 1] = buf; // still receiving data
327     }
328 }
329
330 break;
331
332 case BYTE_DESTUFFING:
333     if(buf == ESCAPE_FLAG) {
334         frame = (unsigned char *)realloc(frame, ++(*size));
335         frame[*size - 1] = FLAG;
336     }
337
338     else if(buf == ESCAPE_ESCAPE) {
339         frame = (unsigned char *)realloc(frame, ++(*size));
340         frame[*size - 1] = ESCAPE_BYTE;
341     }
342
343     else {
344         puts("Character after escape character not
345 recognized"); //can occur if there is an interference
346         errorOnDestuffing = TRUE;
347     }
348
349     current = BCC_OK;
350     break;
351
352 default:
353     break;
354 }
355
356 printf("total size: %d\n", *size);
357 frame = (unsigned char *)realloc(frame, *size - 1);
358 *size = *size - 1;
359
360 printf("Expected trama: %i\n", expectedTrama);
361 printf("Received trama: %i\n", trama);
362
363 if(correctBCC2 && !errorOnDestuffing) {

```

```

364         if(trama == expectedTrama) {
365             if(trama == 0) {
366                 //send RR(Nr = 1)
367                 sendMessage(fd, RR1);
368                 puts("Receiver send RR1");
369             }
370
371             else {
372                 //send RR(Nr = 0)
373                 sendMessage(fd, RR0);
374                 puts("Receiver send RR0");
375             }
376
377             expectedTrama = (expectedTrama + 1) % 2;
378         }
379
380         else { //ao fazermos isto garantimos que a trama recebida e
repetida
381             *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 nao
434     //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 }

```

11.10 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 /**
37  * \brief codifies the Given message into a packet according to the
38  * protocol
39  * @param message message to be sent
40  * @param fileSize total size of the file to be written
41  * @param length total size of the packet to be sent through
42  * llwrite serial port
43  */
44 unsigned char* parseDataPacket(unsigned char *message, off_t
45     fileSize, int *length);
46
47
48 /**
49  * \brief Splits the data into packets that fit into a message (
50  * currently set to 128 bytes)
51  * @param message message containing the whole data
52  * @param index index to start/continue to write the data from
53  * @param packetSize returns the ammount of bytes that can be
54  * written in a single llwrite (128 or less if end of file reached
55  * )
56  * @param fileSize file total size to check how many bytes should
57  * be written
58  * @return returns the packet data that will be sent
59  */
60 unsigned char* splitPacket(unsigned char *message, off_t *index, int
61     *packetSize, off_t fileSize);
62
63
64 /**
65  * \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 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);

```

11.11 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(unsigned
        char) * (*sizeControlPacket));
46
47     if(state == CT.START){
48         packet[0] = CT.START;
49     }
50     else{
51         packet[0] = CT.END;
52     }
53     packet[1] = T1;
54     packet[2] = sizeof(fileSize);
55
56     for(int i = 0; i < packet[2]; i++){
57         packet[3+i] = (fileSize >> (i*8)) & 0xFF;
58     }
59
60     packet[3 + packet[2]] = T2;
61     packet[4 + packet[2]] = sizeFilename;
62
63     for (int i = 0; i < sizeFilename; i++) {

```

```

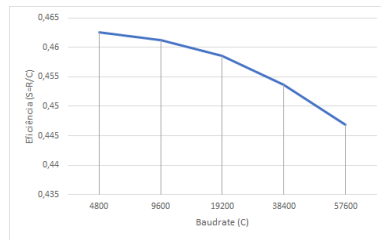
64     packet[5 + packet[2] + i] = filename[i];
65 }
66
67
68     return packet;
69 }
70
71 unsigned char* parseDataPacket(unsigned char *message, off_t
    fileSize, int *length){
72
73     unsigned char *packet = (unsigned char*)malloc(fileSize + 4);
74
75     packet[0] = CONTROL;
76     packet[1] = packetNumber % 255;
77     packet[2] = fileSize / 256;
78     packet[3] = fileSize % 256;
79
80     // Preencher o packet atraves da mensagem
81     for(int i = 0; i < *length; i++){
82         packet[4 + i] = message[i];
83     }
84
85     *length += 4;
86     packetNumber++;
87
88     return packet;
89 }
90
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
110 int checkStart(unsigned char* start, unsigned int *filesize, char *
    name, unsigned int *nameSize){
111
112     int fileSizeBytes;
113
114     // Checking control flag
115     if(start[0] != CT.START || start[1] != T1){
116         puts("checkStart: Error checking CT.START or T1 flags");
117         return -1;

```

```

118     }
119
120     fileSizeBytes = (int)start[2];
121
122     // Getting fileSize
123     for(int i = 0; i < fileSizeBytes; i++){
124         *filesize |= (start[3 + i] << (i*8));
125     }
126
127     if(start[fileSizeBytes + 3] != T2){
128         puts("checkSart: Error checking T2");
129         return -1;
130     }
131
132     // Getting nameSize
133     *nameSize = (unsigned int)start[fileSizeBytes + 4];
134
135     // Getting fileName
136     name = (char *)realloc(name, *nameSize);
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 *
end, int endSize) {
146     int j = 5;
147
148     if(startSize != endSize) {
149         puts("checkEND: Start and End packets have different sizes"
);
150         return 1;
151     }
152     else {
153         if(end[0] == CT_END) {
154             for(int i = 5; i < startSize; i++) {
155                 if(start[i] != end[j]) {
156                     puts("checkEND: Different value between START
and END packets");
157                     return 1;
158                 }
159                 else {
160                     j++;
161                 }
162             }
163             return 0;
164         }
165         else {
166             puts("checkEND: First END packet byte is not CT_END
flag");
167             return 1;
168         }
169     }
170 }

```



(a) label 1

| 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 |

(b) label 2

Figure 1: 2 Figures side by side

```

171
172 unsigned char* assembleDataPacket(unsigned char* message, unsigned
    int messageSize, unsigned int *packetSize){
173
174     *packetSize = messageSize - 4;
175     unsigned char* packet = (unsigned char *)malloc(*packetSize);
176
177     for(int i = 0; i < *packetSize; i++){
178         packet[i] = message[4 + i];
179     }
180
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("File size written: %u\n", fileSize);
189     fclose(file);
190 }

```

11.12 recetor.c

11.13 recetor.c

11.14 Variação do Baudrate

11.15 Variação do tamanho das tramas

11.16 Introdução de atraso de propagação

11.17 Introdução de erros nas tramas

| 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 2: Caption

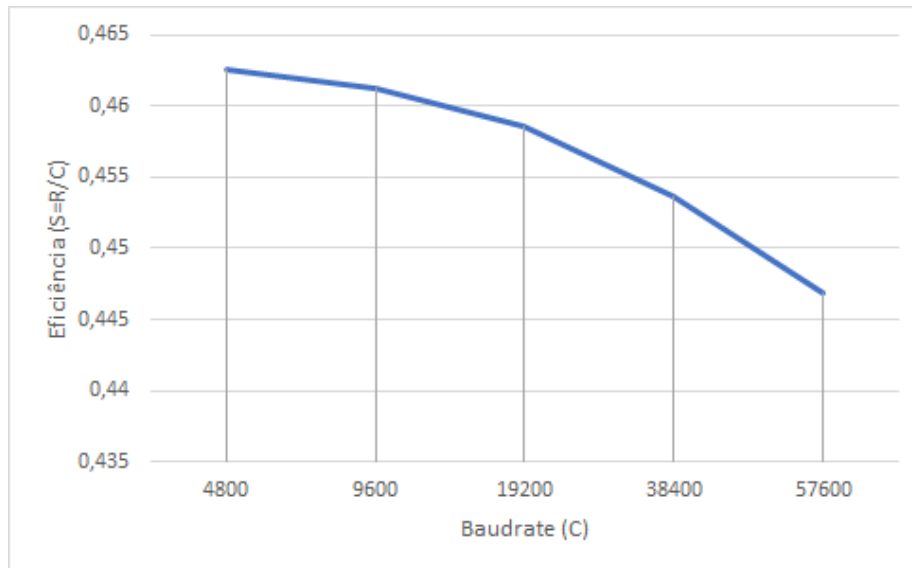


Figure 3: Caption

| 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 4: Caption

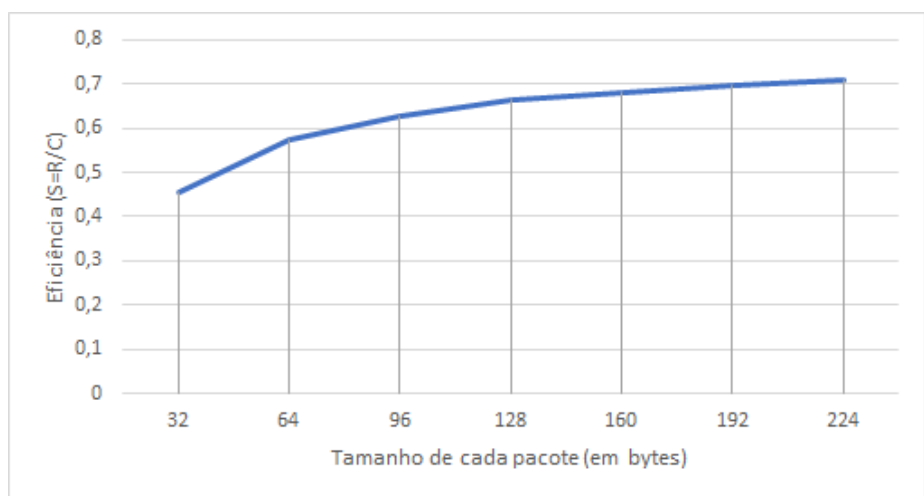


Figure 5: Caption

| 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 6: Caption

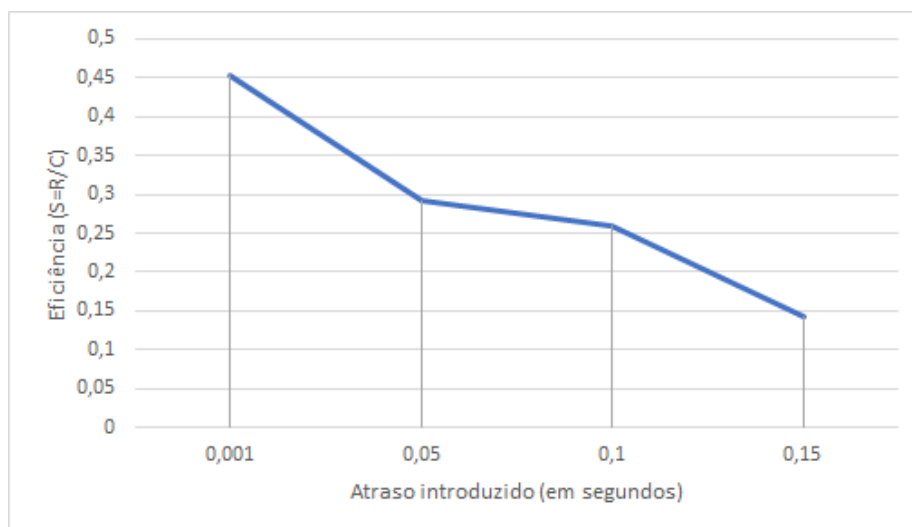


Figure 7: Caption

| 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 8: Caption

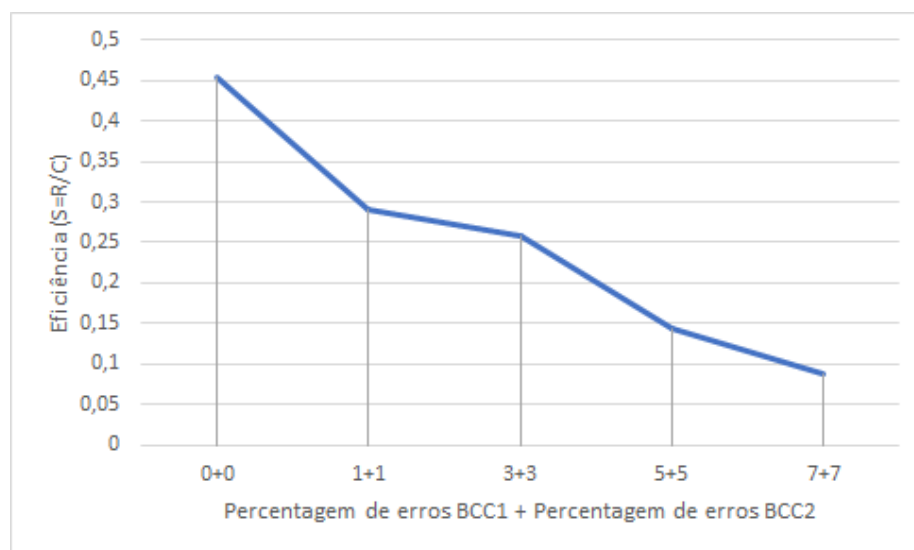


Figure 9: Caption