Protocolo de Ligação de Dados

Relatório

Mestrado Integrado em Engenharia Informática e Computação

Redes de Computadores

3MIEIC03
Bruno Pinto - 201502960
João Santos - 201504013
Vítor Magalhães - 201503447

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

10 de Novembro de 2017

Sumário

Este trabalho tinha como objetivos implementar um protocolo de ligação de dados e testá-lo com uma aplicação simples de transferência de ficheiros, entre dois computadores ligados por uma porta série.

Os objetivos foram cumpridos, ou seja, a aplicação desenvolvida consegue fazer a transferência de ficheiros entre os dois computadores.

1 Introdução

Este trabalho foi desenvolvido com o propósito de testar a transferência de um ficheiro, em ambiente Linux. O mesmo foi feito utilizando C como linguagem de programação, e uma porta série RS-232, em modo não canónico, que liga os dois computadores envolvidos na transferência.

Para garantir o sucesso da transferência, foram as feitas as funções *llopen*, *llwrite*, *llread* e *llclose*

Este documento foi dividido em várias secções por forma a percorrer os seguintes tópicos:

- **Introdução -** Indicação dos objectivos do trabalho e do relatório
- **Arquitetura -** Blocos funcionais e interfaces
- **Estrutura do código -** APIs, principais estruturas de dados, principais funções e sua relação com a arquitetura
- **Casos de uso principais -** Identificação dos casos de uso e sequências de chamada de funções
- **Protocolo de ligação lógica -** Identificação dos principais aspectos funcionais e descrição da estratégia de implementação destes aspectos
- **Protocolo de aplicação -** Identificação dos principais aspectos funcionais e descrição da estratégia de implementação destes aspectos
- **Validação -** Descrição dos testes efectuados
- **Eficiência do protocolo de ligação de dados -** Caraterização estatística da eficiência do protocolo, feita com recurso ao protocolo Stop and Wait
- **Conclusões -** Síntese da informação apresentada nas secções anteriores e reflexão sobre os objectivos de aprendizagem alcançados
- **Anexos -** Código fonte

2 Arquitetura

A aplicação está dividida em vários módulos, cada um responsável por uma entidade da própria aplicação, sendo elas:

- **Camada de aplicação** - Faz a inicialização da sua estrutura de dados, abrindo a porta série, com a configuração adequada.
- **Tramas** - É responsável pela escrita e leitura das tramas, tanto as que são de informação como as que não são.
- **Pacotes** - Assegura a receção e envio dos pacotes de controlo e de dados.
- **Camada de ligação** - Contém as funções *llopen*, *llwrite*, *llread* e *llclose*, ponto central da aplicação.
- ***Stuffing*** - Contém a implementação do mecanismo de tranparência *byte stuffing*.
- **Ficheiro de transferência** - Faz a leitura e escrita do/no ficheiro transferido.

3 Estrutura do código

As estruturas de dados usadas são as que foram sugeridas na especificação do trabalho, ou seja, as da camada de aplicação e camada de ligação.

A camada de aplicação tem como campos o descritor de ficheiro correspondente à porta série e o estado da aplicação na máquina, podendo esta ser o emissor ou o recetor, dependendo do argumento indicado na linha de comandos.

```
typedef struct {
  int fileDescriptor;   /* Descritor correspondente à porta série */
  int status;           /* TRANSMITTER = 0;  RECEIVER = 1 */
} applicationLayer_t;
```

Figura 1: Estrutura da Camada de Aplicação

A camada de ligação tem como campos o caminho da porta série, o *baudrate*, o número de sequência atual da trama, o *timeout*, o número de tentativas de transmissão, para além dos elementos que guardam a última trama genérica recebida e a última trama de dados recebida, após o *destuffing*.

```
typedef struct {
  char port[20];                    /*Porta série /dev/ttySx, x = 0, 1*/
  int baudRate;                     /*Velocidade de transmissão*/
  unsigned int sequenceNumber;      /*Número de sequência da trama*/
  unsigned int timeout;             /*Valor do temporizador: 1 s*/
  unsigned int numTransmissions;    /*Número de tentativas*/
  unsigned char frame[PACKET_SIZE]; /*Trama genérica*/
  unsigned char dataFrame[PACKET_SIZE]; /*Trama de dados*/
} linkLayer_t;
```

Figura 2: Estrutura da Camada de Ligação

As principais funções são as especificadas *llopen*, *llwrite*, *llread* e *llclose*, às quais tomamos a liberdade de fazer algumas alterações, de forma a que fizessem sentido no contexto do restante código desenvolvido. Para além das indicadas, as funções *sendFile*, *receiveFile* são, respetivamente, responsáveis por ler, dividir, enviar e receber o ficheiro e escrevê-lo como ficheiro recebido.

4 Casos de uso principais

A aplicação deste trabalho resulta na transferência de um ficheiro, entre dois computadores ligados por uma porta série. Esta transferência é acompanhada por mensagens apelativas ao processo desta.

Para garantir a igualdade entre os dois computadores, o mesmo programa funciona nos dois dispositivos, havendo apenas a separação entre Emissor e Recetor, como indicado anteriormente, na linha de comandos.

Caso o computador seja o **emissor**, este terá a seguinte sequência:

1 - ***llopen*** **-** começa a conexão, escrevendo um comando SET e espera por um comando UA proveniente do Recetor;

2 - ***sendFile*** **-** utiliza a função writeControlPacket. De seguida, lê o ficheiro a ser enviado e vai progressivamente escrevendo para o Emissor, utilizando a função writeDataPacket;

3 - ***writeControlPacket*** **-** envia uma trama de controlo para o Emissor, utilizando o llwrite;

4 - ***llwrite*** **-** envia uma trama para o Emissor e espera pela resposta deste. Se a resposta for um REJ, tenta de novo. Se for um RR, então termina a função com sucesso.

5 - ***writeDataPacket*** **-** envia uma trama de dados para o Emissor, utilizando o llwrite.

Caso o computador seja o **Recetor**, este terá a seguinte sequência:

1 - ***llopen*** **-** começa a conexão, lendo um comando SET do Emissor e escrevendo um comando UA;

2 - ***receiveFile*** **-** utiliza a função receiveControlPacket. De seguida, vai progressivamente lendo do Emissor, utilizando a função receiveDataPacket e escreve o ficheiro que vai recebendo;

3 - ***receiveControlPacket*** **-** recebe uma trama de controlo do Emissor, utilizando o llread;

4 - ***llread*** **-** recebe uma trama do Emissor e analiza-a, escrevendo para o Emissor a sua resposta. Esta pode ser RR, em caso de sucesso, e REJ em caso de erro;

5 - ***writeDataPacket*** **-** envia uma trama de dados para o Emissor, utilizando o llwrite.

No início, ambas as partes chamam a função ***transferFileInit*** e, no fim, ambas as partes chamam a função ***transferFileClose*** que, respetivamente, abrem e fecham o ficheiro, assim como instanciar a stuct.

5 Protocolo de ligação lógica

A camada de ligação de dados oferece à camada de aplicação funcionalidades que permitem uma transferência eficiente do ficheiro. Esta camada é responsável por estabelecer a ligação de dados através da incialização e abertura da porta série. Esta é utilizada na escrita e leitura de dados - comandos e mensagens, cujo mecanismo de transparência é também assegurado pela *data link layer*.

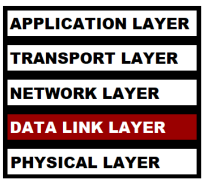

Figura 3: Estrutura da Camada de Ligação

5.1 *llopen*

A **llopen** efetua a conexão entre o emissor e o recetor. O emissor envia o comando **SET** e fica à espera de receber a resposta **UA**, que o recetor envia após a receção do primeiro comando enviado pelo emissor. Todo este procedimento é controlado por um alarme instalado no emissor, com um tempo limite previamente definido, assim como um número limite de tentativas para o estabelecimento da conexão.

5.2 *llread*

A **llread** lê do emissor, utilizando a porta série. Após escrever, analisa a resposta do recetor. Caso seja um **RR** e não tiver ocorrido erros com o número de sequência, o emissor escreve para a porta série essa resposta, troca o número de sequência e termina o ciclo. Se a resposta for um **REJ** escreve outra vez a trama para a porta série.

Tal como a **llopen**, esta função inclui um alarme para controlar possíveis timeouts e tentativas de leitura da resposta.

5.3 *llwrite*

A **llwrite** escreve para o recetor, utilizando a porta série. Ao ler cada trama de informação, chama a função **processDataFrame**, que trata do processamento da trama e devolve a resposta deste. Se esta resposta for um **RR**(Receiver Ready) e não tiver ocorrido erros com o número de sequência, o emissor escreve para a porta série essa resposta, trocando também o número de sequência, terminando o ciclo. Se a resposta for um **REJ**(Reject), escreve esta resposta na porta série.

5.4 *llclose*

A **llclose** efetua a desconexão entre o emissor e o recetor. O emissor envia o comando **DISC** e fica à espera do **DISC** do recetor, após o que envia a resposta **UA**. Todo este procedimento é controlado por alarmes instalados no emissor e no recetor, com um tempo limite previamente definido, assim como um número limite de tentativas para o estabelecimento da conexão.

6 Protocolo de aplicação

A camada da aplicação é a mais alta representada neste trabalho, sendo responsável pelo criação, envio e receção de pacotes - de dados ou controlo - e pelo envio e receção do ficheiro.

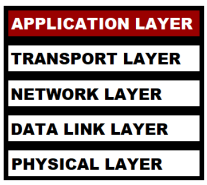

Figura 4: Estrutura da Camada de Aplicação

6.1 Pacotes

Os pacotes da camada de aplicação podem ser de dados ou de controlo - inicial ou final. Esta diferenciação é feita através do primeiro byte do pacote, correspondente ao campo de controlo.

6.1.1 Pacotes de controlo

São os responsáveis por marcar o início e o fim da transmissão do envio e receção do ficheiro.

A ***writeControlPacket*** é chamada pela função ***sendFile*** e é responsável por, dado um campo de controlo, criar os pacotes de controlo e chamar a ***llwrite***, da camada de ligação de dados, que faz o envio. A ***receiveControlPacket*** é chamada pela função ***receiveFile***, cabendo-lhe a tarefa de receber e verificar os pacotes de controlo recebidos, guardando os dados para uso futuro.

```
int writeControlPacket(int controlField, int fd);
int receiveControlPacket(int controlField, int* noBytes, unsigned char** filePath, int fd);
```

Figura 5: Funções relativas aos pacotes de controlo

6.1.2 Pacotes de dados

Estão incluídos nas tramas de informação e transportam fragmentos do ficheiro que está a ser transferido.

A ***writeDataPacket*** é chamada pela função ***sendFile*** e é responsável por, dado um buffer de dados, o número de bytes a transferir e o número de sequência de dados, criar os pacotes de dados e chamar a ***llwrite***, da camada de ligação de dados, que faz o envio. A ***receiveDataPacket*** chama a ***llread***, que faz a receção da trama de informação, é chamada pela função ***receiveFile*** e, após ter feito a verificação dos campos da trama, copia os dados recebidos para um buffer, que posteriormente é escrito para o ficheiro.

```
int writeDataPacket(unsigned char* buffer, int noBytes, int seqNo, int fd);
int receiveDataPacket(unsigned char** buffer, int sequenceNumber, int fd);
```

Figura 6: Funções relativas aos pacotes de dados

6.2 Ficheiro

O envio e receção do ficheiro, de forma eficiente, representam o objetivo principal do trabalho.

A função ***writeFile*** é utilizada pelo emissor e é responsável pelo desencadeamento das funções que permitem o envio do ficheiro. Começa por chamar a ***writeControlPacket*** de forma a enviar o pacote de controlo inicial. De seguida, lê do ficheiro a ser enviado, passando o conteúdo lido para a função ***writeDataPacket***. No final da transferência, a ***writeControlPacket*** é utilizada mais uma vez, para enviar o pacote de controlo final.

A função ***receiveFile*** é utilizada pelo recetor chamando funções que permitem a receção do ficheiro. Começa por chamar a ***receiveControlPacket***, para a receção do pacote de controlo inicial. De seguida, através da função ***receiveDataPacket***, lê da porta série os dados que lhe foram enviados. No final da receção, a ***receiveControlPacket*** é chamada de forma a ser lido o pacote de controlo final.

Durante a transmissão, são apresentadas, tanto no emissor como no recetor, informações relativas ao progresso da transferência.

```
int sendFile(int fd);
int receiveFile(int fd);
```

Figura 7: Funções relativas à transferência e receção do ficheiro

7 Validação

De modo a validar o objetivo do trabalho, foram feitos alguns testes a certos atributos relativos ao mesmo.

Inicialmente, foram feitos testes simples ao envio do ficheiro inicial, *pinguim.gif*. Após confirmar o sucesso da transferência deste, foram testados os ficheiros *penguininv.gif*, *penguin.gif*, *pinguim500k.gif*, *pinguinscaled.gif* e *pinguinsuperscaled.gif*, fornecidos pelos docentes.

Durante a apresentação do trabalho, foi alterado, com sucesso, o *baudrate* para B115200, para a transferência ser mais rápida, assim como o tamanho da trama. Foram feitas simulações de possíveis erros, cortando a ligação entre o emissor e o recetor, temporariamente, pressionando o botão presente na placa. Também foi testada a simulação de erros no BCC2, causando curto-circuito com o fio que se encontra na placa.

Concluiu-se que o trabalhou passou nos testes mencionados, apesar deste não lidar com possíveis erros no BCC1. O sucesso das transferências pode ser confirmado, comparando o ficheiro recebido e as suas propriedades com o ficheiro presente no emissor.

8 Eficiência do protocolo de ligação de dados

De forma a certificar a eficiência do protocolo de ligação de dados, foi executado, utilizando o código original, um mecanismo de Automatic Repeat ReQuest (ARQ), com base no protocolo de Stop and Wait.

O objetivo deste mecanismo é testar a manutenção de erros do trabalho. Quando alguma das tramas é perdida ou contém erros nos dados, o recetor envia uma mensagem **REJ** para o emissor e este envia, de novo, a dita trama. Quando estas tramas são perdidas devido a erros, é indicado ao utilizador a razão desse erro.

Para simular este tipo de erros, procedeu-se a uma verificação com valores diferentes de Frame Error Ratio e Tempos de Propagação diferentes.

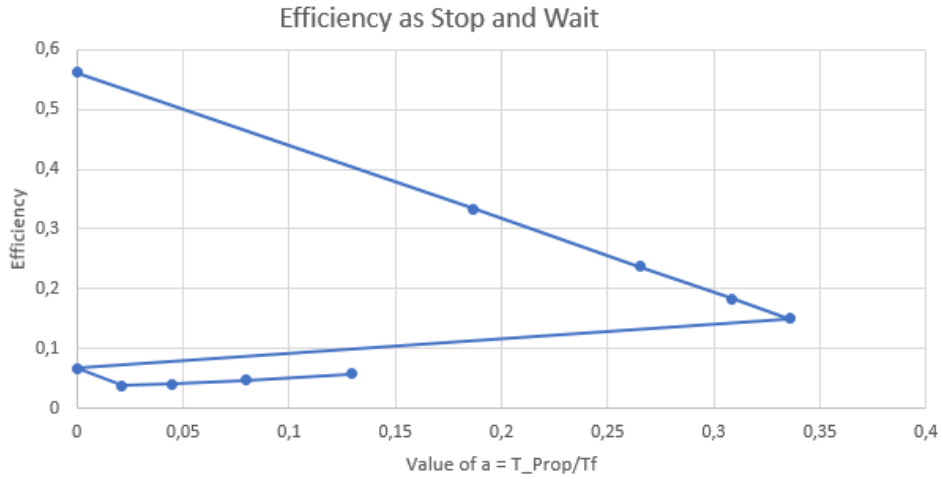

Figura 8: Gráfico da relação Eficiência e "a"

Comparando com o gráfico abordado nas aulas teóricas, conclui-se que o resultado obtido é semelhante àquele que era esperado.

9 Conclusões

Com o desenvolvimento deste trabalho, foi possível compreender o conceito de transferências e conexões entre dois computadores, assim como os vários estágios que têm de ser devidamente implementados e testados. Aquando da sua execução, existe a possibilidade da ocorrência de fenómenos externos que podem levar a falhas no resultado final do programa. Tendo isso em mente, teve-se especial atenção na manutenção de erros, abrangendo diferentes propriedades de ficheiro e portas série utilizadas.

Devido à necessidade de monitorizar o desempenho do programa, foram implementados cálculos estatísticos referentes à eficiência do protocolo de ligação de dados.

Assim foi criado um programa capaz de transferir o ficheiro com sucesso e verificar possíveis erros na transferência e/ou na conexão .

Anexos

alarm.c

```c
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <termios.h>
#include <stdlib.h>
#include "applicationLayer.h"
#include "linkLayer.h"

int alarmFlag = 0;
int fileDescriptor = 0;

void setVMIN(int noChars)
{
    struct termios oldtio;

    /* save current port settings */
    if (tcgetattr(appL->fileDescriptor, &oldtio) == -1) {
        perror("tcgetattr");
        exit(-1);
    }

    oldtio.c_cc[VMIN] = noChars;

    if (tcflush(appL->fileDescriptor, TCIFLUSH) == -1) {
        perror("tcflush");
        exit(-1);
    }

    if (tcsetattr(appL->fileDescriptor, TCSANOW, &oldtio) == -1)
    ↪ {
        perror("tcsetattr");
        exit(-1);
    }
}

void alarmHandler()
{
    printf("Alarm!\n");

    alarmFlag = 1;

    setVMIN(0);
}

void setAlarm(int fd)
{
    (void)signal(SIGALRM, alarmHandler);

    alarmFlag = 0;

    setVMIN(1);

    alarm(linkL->timeout);
```

```
53
54      fileDescriptor = fd;
55  }
56
57  void stopAlarm(int fd)
58  {
59      (void)signal(SIGALRM, NULL);
60
61      alarm(0);
62
63      fileDescriptor = fd;
64  }
```

alarm.h

```
1   #pragma once
2
3   extern int alarmFlag;
4   extern int fileDescriptor;
5
6   /**
7    * Handles the alarm by printing a message
8   */
9   void alarmHandler();
10
11  /**
12    * Sets the alarm
13     @param fd É PARA REMOVER
14  */
15  void setAlarm(int fd);
16
17  /**
18    * Stops the alarm
19     @param fd É PARA REMOVER
20  */
21  void stopAlarm(int fd);
22
23  /**
24    * Sets a new VMIN on termios
25     @param noChars new value for VMIN
26  */
27  void setVMIN(int noChars);
```

applicationLayer.c

```
1   #include <stdlib.h>
2   #include <sys/types.h>
3   #include <sys/stat.h>
4   #include <fcntl.h>
5   #include <stdio.h>
6   #include <termios.h>
7   #include <strings.h>
8   #include <unistd.h>
9   #include <string.h>
10  #include "transferFile.h"
11  #include "applicationLayer.h"
12  #include "linkLayer.h"
```

```c
#include "configs.h"
#include "definitions.h"
#include "alarm.h"

applicationLayer_t* appL;
struct termios oldtio, newtio;

int applicationLayerInit(int status)
{
    appL = (applicationLayer_t*)malloc(sizeof(applicationLayer_t));

    if ((appL->fileDescriptor = open(linkL->port, O_RDWR | O_NOCTTY)) < 0) {
        printf("Error on oppening serial port!\n");
        exit(1);
    }
    else {
        printf("Serial port opened!\n");
    }

    appL->status = status;

    if (appL->status == TRANSMITTER) {
        printf("\nWhat is the file location?\t");
        scanf("%s", FILE_PATH);
        printf("\n");
    }

    transferFileInit(appL->status);

    return 0;
}

int saveAndSetTermios()
{
    /* save current port settings */
    if (tcgetattr(appL->fileDescriptor, &oldtio) == -1) {
        perror("tcgetattr");
        return 1;
    }

    bzero(&newtio, sizeof(newtio));
    newtio.c_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;
    newtio.c_iflag = IGNPAR;
    newtio.c_oflag = OPOST;

    /* set input mode (non-canonical, no echo,...) */
    newtio.c_lflag = 0;
    newtio.c_cc[VTIME] = 0; /* inter-character timer unused */
    newtio.c_cc[VMIN] = 1; /* blocking read until a char is received */

    if (tcflush(appL->fileDescriptor, TCIFLUSH) == -1) {
        perror("tcflush");
        return 1;
    }
```

```
67
68      if (tcsetattr(appL->fileDescriptor, TCSANOW, &newtio) == -1)
        ↪  {
69          perror("tcsetattr");
70          return 1;
71      }
72
73      return 0;
74  }
75
76  int closeSerialPort(int fd)
77  {
78      if (tcsetattr(fd, TCSANOW, &oldtio) == -1) {
79          perror("tcsetattr");
80          return 1;
81      }
82
83      close(fd);
84
85      return 0;
86  }
```

applicationLayer.h

```
1   #pragma once
2   #include <stdio.h>
3   #include "linkLayer.h"
4
5   /**
6           Contains the serial port file descriptor and if user is
    ↪  TRANSMITTER or RECEIVER
7   */
8   typedef struct {
9       int fileDescriptor; /* Descritor correspondente à porta série
        ↪  */
10      int status; /* TRANSMITTER = 0;  RECEIVER = 1 */
11  } applicationLayer_t;
12
13  extern applicationLayer_t* appL;
14
15  /**
16          * Initiates an application layer struct
17  */
18  int applicationLayerInit(int status);
19
20  /**
21          * Closes serial port
22          @param fd Serial Port's file descriptor
23  */
24  int closeSerialPort(int fd);
25
26  /**
27          * Sets a new termios struct
28  */
29  int saveAndSetTermios();
```

configs.h

```
#pragma once

#define TIMEOUT 5
#define BAUDRATE B38400
#define NUM_TRANSMISSIONS 3
#define PACKET_SIZE 428
#define NO_TRIES 5
```

definitions.h

```
#pragma once
/**
        User Status
*/
#define TRANSMITTER 0
#define RECEIVER 1

#define CTRL_START 2
#define CTRL_END 3

#define COMMAND_SIZE 5
#define DATA_SIZE 6

#define DATA_BYTE 1
#define START_BYTE 2
#define END_BYTE 3

#define FLAG 0x7E //First and last byte

#define ADDR_S 0x03 //Sender perspective: commands sent by Sender and sent by Receiver(or received by Sender)
#define ADDR_R 0x01 //Receiver perspective: commands sent by Receiver and sent by Sender(or received by Receiver)

#define CTRL_SET 0x03 //set up
#define CTRL_DISC 0x0B //disconnect
#define CTRL_UA 0x07 //acknowledgement
#define CTRL_RR 0x05 //receiver ready
#define CTRL_REJ 0x01 //reject

#define ESC_BYTE 0x7D

#define FILE_SIZE 0
#define FILE_NAME 1

#define STR_SIZE 128

#define BIT(n) (1 << (n))
```

handleFrames.c

```
#include "handleFrames.h"
#include "definitions.h"
#include "linkLayer.h"
#include "applicationLayer.h"
```

```c
#include "stuffing.h"
#include "alarm.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <termios.h>

int writeNonDataFrame(Frame frame, int fd)
{
    unsigned char buf[COMMAND_SIZE];

    buf[0] = FLAG;
    buf[1] = ADDR_S;

    switch (frame) {
    case SET:
        buf[2] = CTRL_SET;
        break;
    case DISC:
        buf[2] = CTRL_DISC;
        break;
    case UA:
        buf[2] = CTRL_UA;
        break;
    case RR:
        buf[2] = CTRL_RR | (linkL->sequenceNumber << 5);
        break;
    case REJ:
        buf[2] = CTRL_REJ | (linkL->sequenceNumber << 5);
        break;
    default:
        break;
    }

    buf[3] = buf[1] ^ buf[2];
    buf[4] = FLAG;

    tcflush(appL->fileDescriptor, TCOFLUSH);

    int res;

    if ((res = write(fd, &buf, 5)) != sizeof(buf)) {
        printf("Error on writting!\n");
        return -1;
    }

    return 0;
}

int writeDataFrame(unsigned char* data, unsigned int length, int
↪ fd)
{
    unsigned char* frame = malloc(1024);
    int size = length + DATA_SIZE;
    unsigned char bcc2 = 0;
    int dataInd = 0;
```

```
61
62      frame[0] = FLAG;
63      frame[1] = ADDR_S;
64      frame[2] = linkL->sequenceNumber << 5;
65      frame[3] = frame[1] ^ frame[2];
66
67      memcpy(&frame[4], data, length);
68
69      int counter = 0;
70
71      for (dataInd = 0; dataInd < length; dataInd++) {
72          bcc2 ^= data[dataInd];
73          counter++;
74      }
75
76      frame[4 + length] = bcc2;
77      frame[5 + length] = FLAG;
78
79      frame = stuffing(frame, &size);
80
81      int res = 0;
82
83      if ((res = write(fd, frame, size)) != size) {
84          printf("Error on writing data frame!\n");
85          exit(1);
86      }
87
88      return 0;
89  }
90
91  int receiveFrame(int* fSize, int fd)
92  {
93      unsigned char c;
94      int res, ind = 0;
95      ReceivingState rState = 0;
96
97      while (alarmFlag != 1 && rState != STOP) {
98          res = read(fd, &c, 1);
99
100         if (res > 0) {
101
102             switch (rState) {
103             case START:
104                 if (c == FLAG) {
105                     linkL->frame[ind++] = c;
106                     rState++;
107                 }
108                 break;
109             case FLAG_RCV:
110                 if (c == ADDR_S || c == ADDR_R) {
111                     linkL->frame[ind++] = c;
112                     rState++;
113                 }
114                 else if (c != FLAG) {
115                     rState = START;
116                     ind = 0;
117                 }
```

```
118                 break;
119             case A_RCV:
120                 if (c != FLAG) {
121                     linkL->frame[ind++] = c;
122                     rState++;
123                 }
124                 else if (c == FLAG) {
125                     rState = FLAG_RCV;
126                     ind = 1;
127                 }
128                 else {
129                     rState = START;
130                     ind = 0;
131                 }
132                 break;
133             case C_RCV:
134                 if (c == (linkL->frame[1] ^ linkL->frame[2])) {
135                     linkL->frame[ind++] = c;
136                     rState++;
137                 }
138                 else {
139                     if (c == FLAG) {
140                         rState = FLAG_RCV;
141                         ind = 1;
142                     }
143                     else {
144                         rState = START;
145                         ind = 0;
146                     }
147                 }
148                 break;
149             case BCC_OK:
150                 if (c == FLAG) {
151                     linkL->frame[ind++] = c;
152                     rState++;
153                 }
154                 else
155                     linkL->frame[ind++] = c;
156                 break;
157             case STOP:
158                 break;
159             default:
160                 break;
161             }
162         }
163     }
164
165     (*fSize) = ind;
166
167     if (ind > 5)
168         return DATA;
169     else
170         return NON_DATA;
171 }
172
173 void processDataFrame(FrameResponse* fResp, int size)
174 {
```

```
    unsigned char bcc2 = 0;
    int dataInd;
    (*fResp) = 0;

    int destuffedSize = size;

    unsigned char* destuffed;

    destuffed = destuffing(linkL->frame, &destuffedSize);

    strcpy((char*)linkL->frame, (char*)destuffed);

    int counter = 0;

    for (dataInd = 4; dataInd < (destuffedSize - 2); dataInd++) {
        bcc2 ^= destuffed[dataInd];
        counter++;
    }

    if (destuffed[destuffedSize - 2] != bcc2) {
        printf("Error on BCC2!\n");
        (*fResp) = RESP_REJ;
    }

    if (*fResp == 0)
        (*fResp) = RESP_RR;

    memcpy(linkL->dataFrame, destuffed, size);
}
```

handleFrames.h

```
#pragma once

/* Reception state machine */
typedef enum {
    START,
    FLAG_RCV,
    A_RCV,
    C_RCV,
    BCC_OK,
    STOP
} ReceivingState;

/* Frame responses */
typedef enum {
    RESP_RR,
    RESP_REJ
} FrameResponse;

/* Non data frames */
typedef enum {
    SET,
    DISC,
    UA,
    RR,
    REJ
```

```c
} Frame;

/* Frame type */
typedef enum {
    DATA,
    NON_DATA
} FrameType;

/**
* Writes a data frame to the serial port
* @param data Data to be written
* @param length Length of data
* @param fd Serial port file descriptor
*/
int writeDataFrame(unsigned char* data, unsigned int length, int
↪  fd);

/**
* Receives a frame from the serial port
* @param fSize Returns the frame size
* @param fd Serial port file descriptor
*/
int receiveFrame(int* fSize, int fd);

/**
* Writes a non data frame to the serial port
* @param frame Frame type to be written
* @param fd Serial port file descriptor
*/
int writeNonDataFrame(Frame frame, int fd);

/**
* Processes a data frame received from the serial port
* @param fResp Returns the frame response
* @param size Frame size
*/
void processDataFrame(FrameResponse* fResp, int size);
```

handlePackets.c

```c
#include <string.h>
#include <stdlib.h>
#include "handlePackets.h"
#include "linkLayer.h"
#include "transferFile.h"
#include "definitions.h"

int writeControlPacket(int controlField, int fd)
{
    unsigned char fileSize[14]; /* 10.7KB = 10 700B | log2(10
    ↪  700)~=14 */
    sprintf((char*)fileSize, "%d", traF->fileSize);

    int ctrlPkSize = 5 + strlen((char*)fileSize) +
    ↪  strlen(FILE_PATH); /* 5 bytes
    from C, T1, L1, T2 and L2 */

```

```
16      unsigned char controlPacket[ctrlPkSize];
17
18      controlPacket[0] = controlField;
19      controlPacket[1] = FILE_SIZE;
20      controlPacket[2] = strlen((char*)fileSize);
21
22      int index = 3;
23      int k;
24
25      for (k = 0; k < strlen((char*)fileSize); k++, index++)
26          controlPacket[index] = fileSize[k];
27
28      controlPacket[index++] = FILE_NAME;
29      controlPacket[index++] = strlen(FILE_PATH);
30
31      for (k = 0; k < strlen(FILE_PATH); k++, index++)
32          controlPacket[index] = FILE_PATH[k];
33
34      if (llwrite(controlPacket, ctrlPkSize, fd) != 0) {
35          printf("Error on llwrite!\n");
36          return 1;
37      }
38
39      return 0;
40  }
41
42  int writeDataPacket(unsigned char* buffer, int noBytes, int
    ↪  seqNo, int fd)
43  {
44      int dataPkSize = noBytes + 4; /* 4 bytes from C, N, L2 and L1
        ↪  */
45
46      unsigned char dataPacket[dataPkSize];
47
48      dataPacket[0] = DATA_BYTE;
49      dataPacket[1] = seqNo;
50
51      /* K = 256 * dataPacket[2] + dataPacket[3] */
52      dataPacket[2] = noBytes / 256;
53      dataPacket[3] = noBytes % 256;
54      memcpy(&dataPacket[4], buffer, noBytes);
55
56      if (llwrite(dataPacket, dataPkSize, fd) != 0) {
57          printf("Error on llwrite!\n");
58          return 1;
59      }
60
61      return 0;
62  }
63
64  int receiveControlPacket(int controlField, int* noBytes, unsigned
    ↪  char** filePath, int fd)
65  {
66      unsigned char* controlPacket;
67
68      if (llread(&controlPacket, fd)) {
```

```
69          printf("Error on receiving control packet at
            ↪  llread()!\n");
70          return 1;
71      }
72
73      if ((controlPacket[0]) != controlField) {
74          printf("Unexpected control field!\n");
75          return 1;
76      }
77
78      if ((controlPacket[1]) != FILE_SIZE) {
79          printf("Unexpected parameter!\n");
80          return 1;
81      }
82
83      int lengthSize = (controlPacket[2]);
84      int i, valueIndex = 3;
85      unsigned char fileSize[STR_SIZE];
86
87      for (i = 0; i < lengthSize; i++)
88          fileSize[i] = controlPacket[valueIndex++];
89
90      fileSize[valueIndex - 3] = '\0';
91      (*noBytes) = atoi((char*)fileSize);
92
93      if ((controlPacket[valueIndex++]) != FILE_NAME)
94          printf("Unexpected parameter!\n");
95
96      int lengthPath = (controlPacket[valueIndex++]);
97      unsigned char path[STR_SIZE];
98
99      for (i = 0; i < lengthPath; i++)
100         path[i] = controlPacket[valueIndex++];
101
102     path[i] = '\0';
103
104     strcpy((char*)(*filePath), (char*)path);
105
106     return 0;
107 }
108
109 int receiveDataPacket(unsigned char** buffer, int sequenceNumber,
    ↪  int fd)
110 {
111     unsigned char* dataPacket;
112     int read;
113
114     if (llread(&dataPacket, fd)) {
115         printf("Error on receiving data packet at llread()!\n");
116         exit(1);
117     }
118
119     int controlField = dataPacket[0];
120     int seqNo = dataPacket[1];
121
122     if (controlField != DATA_BYTE) {
123         printf("Unexpected control field!\n");
```

```
124             exit(1);
125         }
126 
127         if (seqNo != sequenceNumber) {
128             printf("Unexpected sequence number!\n");
129             exit(1);
130         }
131 
132         int l2 = dataPacket[2], l1 = dataPacket[3];
133         read = 256 * l2 + l1;
134 
135         memcpy((*buffer), &dataPacket[4], read);
136         free(dataPacket);
137 
138         return read;
139     }
```

handlePackets.h

```
1   #pragma once
2   
3   /**
4           * Writes Control Packet
5           @param controlField Start Byte or End Byte
6           @param fd Serial port's file descriptor
7   */
8   int writeControlPacket(int controlField, int fd);
9   
10  /**
11          * Writes Data Packet
12          @param buffer Data to be sent
13          @param noBytes Number of bytes to be sent
14          @param seqNo Sequence number of the Data
15          @param fd Serial port's file descriptor
16  */
17  int writeDataPacket(unsigned char* buffer, int noBytes, int
    ↪  seqNo, int fd);
18  
19  /**
20          * Receives Control Packet
21          @param controlField Start Byte or End Byte
22          @param noBytes Number of bytes to be received
23          @param filePath Path of file to be received
24          @param fd Serial port's file descriptor
25  */
26  int receiveControlPacket(int controlField, int* noBytes, unsigned
    ↪  char** filePath, int fd);
27  
28  /**
29          * Receives Data Packet
30          @param buffer Data to be sent
31          @param seqNo Sequence number of the Data
32          @param fd Serial port's file descriptor
33  */
34  int receiveDataPacket(unsigned char** buffer, int sequenceNumber,
    ↪  int fd);
```

linkLayer.c

```c
#include <stdlib.h>
#include <string.h>
#include <termios.h>
#include <unistd.h>
#include <errno.h>
#include "alarm.h"
#include "definitions.h"
#include "configs.h"
#include "stuffing.h"
#include "handlePackets.h"
#include "linkLayer.h"
#include "applicationLayer.h"
#include "transferFile.h"
#include "handleFrames.h"

linkLayer_t* linkL;

int linkLayerInit(char* port, int status)
{
    linkL = (linkLayer_t*)malloc(sizeof(linkLayer_t));

    strcpy(linkL->port, port);

    linkL->timeout = TIMEOUT;
    linkL->baudRate = BAUDRATE; /*Velocidade de transmissão*/
    linkL->sequenceNumber = 0; /*Número de sequência da trama: 0,
    ↪  1*/
    linkL->numTransmissions = NUM_TRANSMISSIONS; /*Número de
    ↪  tentativas em caso de falha*/

    applicationLayerInit(status);
    int fd = appL->fileDescriptor;

    saveAndSetTermios();

    if (llopen(fd)) {
        printf("Error in llopen!\n");
        exit(1);
    }

    switch (appL->status) {
    case TRANSMITTER:
        sendFile(fd);
        break;
    case RECEIVER:
        receiveFile(fd);
        break;
    default:
        exit(1);
    }

    if (llclose(fd)) {
        printf("Error in llclose!\n");
        exit(1);
    }
```

```
54
55      closeSerialPort(fd);
56
57      return 0;
58  }
59
60  int llopen(int fd)
61  {
62      int alarmCounter = 0;
63      int fSize;
64
65      switch (appL->status) {
66      case TRANSMITTER:
67          while (alarmCounter < NO_TRIES) {
68              if (alarmCounter == 0 || alarmFlag == 1) {
69                  setAlarm(fd);
70                  writeNonDataFrame(SET, fd);
71                  alarmFlag = 0;
72                  alarmCounter++;
73              }
74
75              receiveFrame(&fSize, fd);
76
77              if (linkL->frame[2] == CTRL_UA) {
78                  break;
79              }
80          }
81          stopAlarm(fd);
82          if (alarmCounter < NO_TRIES)
83              printf("Connection done!\n");
84          else {
85              printf("Connection couldn't be done!\n");
86              return 1;
87          }
88          break;
89
90      case RECEIVER:
91          receiveFrame(&fSize, fd);
92
93          if (linkL->frame[2] == CTRL_SET) {
94              writeNonDataFrame(UA, fd);
95              printf("Connection done!\n");
96          }
97          else {
98              printf("Connection couldn't be done!\n");
99              return 1;
100         }
101         break;
102     }
103
104     return 0;
105 }
106
107 int llwrite(unsigned char* buffer, int length, int fd)
108 {
109     int alarmCounter = 0;
110     int fSize;
```

```
111
112     while (alarmCounter < NO_TRIES) {
113         if (alarmCounter == 0 || alarmFlag == 1) {
114             setAlarm(fd);
115             writeDataFrame(buffer, length, fd);
116             alarmFlag = 0;
117             alarmCounter++;
118         }
119
120         receiveFrame(&fSize, fd);
121
122         if (linkL->frame[2] == (CTRL_RR | (linkL->sequenceNumber
            ↪  << 5))) {
123             stopAlarm(fd);
124             linkL->sequenceNumber = !linkL->sequenceNumber;
125             break;
126         }
127         else if (linkL->frame[2] == (CTRL_REJ |
            ↪  (linkL->sequenceNumber << 5))) {
128         }
129     }
130
131     if (alarmCounter < NO_TRIES) {
132         return 0;
133     }
134     else {
135         stopAlarm(fd);
136         printf("Couldn't write!\n");
137         return 1;
138     }
139
140     return 1;
141 }
142
143 int llread(unsigned char** buffer, int fd)
144 {
145     int fSize, dataSize, answered = 0;
146     FrameResponse fResp = 0;
147
148     while (answered == 0) {
149         if (receiveFrame(&fSize, fd) == DATA) {
150             processDataFrame(&fResp, fSize);
151         }
152
153         if (fResp == RESP_RR && ((linkL->frame[2] >> 5) & BIT(0))
            ↪  == linkL->sequenceNumber) {
154             writeNonDataFrame(RR, fd);
155             linkL->sequenceNumber = !linkL->sequenceNumber;
156             dataSize = fSize - DATA_SIZE;
157             *buffer = malloc(dataSize);
158             memcpy(*buffer, &linkL->dataFrame[4], dataSize);
159             answered = 1;
160         }
161         else if (fResp == RESP_REJ) {
162             linkL->sequenceNumber = ((linkL->frame[2] >> 5) &
                ↪  BIT(0));
163             writeNonDataFrame(REJ, fd);
```

```
164             }
165         }
166 
167         return 0;
168 }
169 
170 int llclose(int fd)
171 {
172         int alarmCounter = 0;
173         int discReceived = 0;
174         int fSize;
175 
176         switch (appL->status) {
177         case TRANSMITTER:
178             while (alarmCounter < NO_TRIES) {
179 
180                 if (alarmCounter == 0 || alarmFlag == 1) {
181                     setAlarm(fd);
182                     writeNonDataFrame(DISC, fd);
183                     alarmFlag = 0;
184                     alarmCounter++;
185                 }
186 
187                 receiveFrame(&fSize, fd);
188 
189                 if (linkL->frame[2] == CTRL_DISC) {
190                     writeNonDataFrame(UA, fd);
191                     break;
192                 }
193             }
194 
195             stopAlarm(fd);
196 
197             if (alarmCounter < NO_TRIES) {
198                 printf("Disconnection done!\n");
199                 return 0;
200             }
201             else {
202                 printf("Disconnection couldn't be done!\n");
203                 return 1;
204             }
205             break;
206 
207         case RECEIVER:
208             while (alarmCounter < NO_TRIES) {
209                 receiveFrame(&fSize, fd);
210 
211                 if (linkL->frame[2] == CTRL_DISC) {
212                     discReceived = 1;
213                 }
214 
215                 if (discReceived && (alarmCounter == 0 || alarmFlag
                    ↪  == 1)) {
216                     setAlarm(fd);
217                     writeNonDataFrame(DISC, fd);
218                     alarmFlag = 0;
219                     alarmCounter++;
```

```
            }

            sleep(1);
            receiveFrame(&fSize, fd);

            if (linkL->frame[2] == CTRL_UA) {
                break;
            }
        }

        stopAlarm(fd);

        if (alarmCounter < NO_TRIES) {
            printf("Disconnection done!\n");
            return 0;
        }
        else {
            printf("Disconnection couldn't be done!\n");
            return 1;
        }
        break;
    }

    return 0;
}
```

linkLayer.h

```
#pragma once
#include "configs.h"

typedef struct {
    char port[20]; /*Dispositivo /dev/ttySx, x = 0, 1*/
    int baudRate; /*Velocidade de transmissão*/
    unsigned int sequenceNumber; /*Número de sequência da trama: 0, 1*/
    unsigned int timeout; /*Valor do temporizador*/
    unsigned int numTransmissions; /*Número de tentativas em caso de falha*/
    unsigned char frame[PACKET_SIZE]; /*Trama*/
    unsigned char dataFrame[PACKET_SIZE]; /*Trama de dados*/
} linkLayer_t;

extern linkLayer_t* linkL;

/**
    * Initiates a new Link Layer struct and starts the connection
    @param port serial port used
    @param status indicates if the user is the Transmitter or Receiver
*/
int linkLayerInit(char* port, int status);

/**
    * Establishes the connection between the two computers
    @param fd serial port's file descriptor
*/
```

```
int llopen(int fd);

/**
    * Transmitter sends frames to the Receiver
    @param buffer data packet
    @param length data packet size
    @param fd serial port's file descriptor
*/
int llwrite(unsigned char* buffer, int length, int fd);

/**
    * Receiver reads frames from the Transmitter
    @param buffer data packet
    @param fd serial port's file descriptor
*/
int llread(unsigned char** buffer, int fd);

/**
    * Closes the connection
*/
int llclose(int fd);
```

main.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include <stdlib.h>
#include "linkLayer.h"

/**
        * Verifies command line arguments and begins program
*/
int main(int argc, char** argv)
{

    if ((argc < 3) || ((strcmp("/dev/ttyS0", argv[1]) != 0) &&
     ↪  (strcmp("/dev/ttyS1", argv[1]) != 0))) {
        printf("Usage:\tSerialPort ProgramStatus\n\tex:
         ↪  /dev/ttyS1 1\n");
        exit(1);
    }

    char port[20];
    strcpy(port, argv[1]);
    int status = atoi(argv[2]);

    linkLayerInit(port, status);

    return 0;
}
```

stuffing.c

```c
#include "stuffing.h"
#include "definitions.h"
#include <stdlib.h>
#include <string.h>

unsigned char* stuffing(unsigned char* buf, int* size)
{
    unsigned char* stuffed = malloc(1024);
    int stuffedSize = (*size);

    stuffed[0] = buf[0];

    int i, j = 1;

    for (i = 1; i < ((*size) - 1); i++, j++) {
        if (buf[i] == FLAG || buf[i] == ESC_BYTE) {
            stuffedSize++;
            stuffed[j] = ESC_BYTE;
            j++;
            stuffed[j] = buf[i] ^ 0x20;
        }
        else
            stuffed[j] = buf[i];
    }

    stuffed[j] = buf[i];

    (*size) = stuffedSize;

    return stuffed;
}

unsigned char* destuffing(unsigned char* buf, int* size)
{
    unsigned char* destuffed = malloc(1024);

    int destuffedSize = (*size);

    int i, j = 4;

    for (i = 0; i < 4; i++)
        destuffed[i] = buf[i];

    i = 0;

    for (i = 4; i < ((*size) - 1); i++) {
        if (buf[i] == FLAG || buf[i] == ESC_BYTE) {
            i++;
            destuffed[j] = buf[i] ^ 0x20;
            destuffedSize--;
        }
        else {
            destuffed[j] = buf[i];
        }
        j++;
```

```
    }

    for (i = (*size) - 1; i < (*size); i++) {
        destuffed[j] = buf[i];
        j++;
    }

    (*size) = destuffedSize;

    return destuffed;
}
```

stuffing.h

```
#pragma once

/**
        Handles stuffing technique
        @param buf Data to be stuffed
        @param size Data buffer size
*/
unsigned char* stuffing(unsigned char* buf, int* size);

/**
        Handles destuffing technique
        @param buf Data to be stuffed
        @param size Data buffer size
*/
unsigned char* destuffing(unsigned char* buf, int* size);
```

transferFile.c

```
#include "definitions.h"
#include "configs.h"
#include "transferFile.h"
#include "handlePackets.h"
#include <stdlib.h>

transferFile_t* traF;
char FILE_PATH[50];

int transferFileInit(int status)
{
    struct stat st;

    traF = (transferFile_t*)malloc(sizeof(transferFile_t));

    if (status == TRANSMITTER) {
        if (!(traF->file = fopen(FILE_PATH, "rb"))) {
            printf("Unable to open file!\n");
            exit(1);
        }

        if (stat(FILE_PATH, &st) == 0)
            traF->fileSize = st.st_size;
        else {
            printf("Unable to get file size!\n");
```

```
26              exit(1);
27          }
28      }
29
30      printf("Transfer file opened!\n");
31
32      return 0;
33  }
34
35  void transferFileClose()
36  {
37      if (fclose(traF->file) < 0) {
38          printf("Unable to close transfer file!\n");
39          exit(1);
40      }
41
42      printf("Transfer file closed!\n");
43  }
44
45  int sendFile(int fd)
46  {
47      unsigned char* packetBuffer = malloc(PACKET_SIZE *
        ↪  sizeof(unsigned char));
48      int read, seqNo = 0, noBytes = 0;
49
50      if (writeControlPacket(CTRL_START, fd)) {
51          printf("Error on writing control packet in sendFile!\n");
52          exit(1);
53      }
54      else {
55          printf("Start control packet written!\n\n");
56      }
57
58      while ((read = fread(packetBuffer, sizeof(unsigned char),
        ↪  PACKET_SIZE, traF->file)) > 0) {
59          printf("%d bytes read from file!\n", read);
60
61          if (writeDataPacket(packetBuffer, read, (seqNo % 255),
            ↪  fd)) { //seqNo is module 255
62              printf("Error on writing data packet in
                ↪  sendFile!\n");
63              exit(1);
64          }
65
66          noBytes += read;
67
68          printf("Progress = %.2f\n", (noBytes/traF->fileSize *
            ↪  100));
69          printf("seqNo sent = %d\n\n", (seqNo % 255));
70
71          seqNo++;
72      }
73
74      transferFileClose();
75
76      if (writeControlPacket(CTRL_END, fd)) {
77          printf("Error on writing control packet in sendFile!\n");
```

```
78          exit(1);
79      }
80      else {
81          printf("End control packet written!\n");
82      }
83
84      return 0;
85  }
86
87  int receiveFile(int fd)
88  {
89      int fileSize;
90      unsigned char* filePath = (unsigned char*)malloc(100);
91
92      if (receiveControlPacket(START_BYTE, &fileSize, &filePath,
        ↪  fd)) {
93          printf("Error on receiving control packet in
            ↪  receiveFile!\n");
94          exit(1);
95      }
96      else {
97          printf("Start control packet received!\n\n");
98      }
99
100     if (!(traF->file = fopen((char*)filePath, "wb"))) {
101         printf("Unable to open file!\n");
102         exit(1);
103     }
104
105     int read, noBytes = 0, seqNo = 0, written = 0;
106     unsigned char* buffer = malloc(PACKET_SIZE * sizeof(char));
107
108     while (noBytes < fileSize) {
109         if ((read = receiveDataPacket(&buffer, (seqNo % 255),
            ↪  fd)) < 0)
110             exit(1);
111
112         noBytes += read;
113         printf("seqNo received = %d\n", (seqNo % 255));
114         printf("Progress = %.2f\n", (noBytes/fileSize * 100));
115
116         if ((written = fwrite(buffer, sizeof(char), read,
            ↪  traF->file)) > 0) {
117             printf("%d bytes written to file!\n\n", written);
118         }
119         else {
120             return 1;
121         }
122
123         seqNo++;
124     }
125
126     transferFileClose();
127
128     if (receiveControlPacket(END_BYTE, &fileSize, &filePath, fd))
        ↪  {
```

```
129         printf("Error on receiving control packet in
            ↪  receiveFile!\n");
130         exit(1);
131     }
132     else {
133         printf("End control packet received!\n");
134     }
135 
136     return 0;
137 }
```

transferFile.h

```
1  #pragma once
2  
3  #include <sys/stat.h>
4  #include <stdio.h>
5  
6  typedef struct {
7      FILE* file;
8      int fileSize;
9  } transferFile_t;
10 
11 extern transferFile_t* traF;
12 
13 extern char FILE_PATH[50];
14 
15 /**
16         Opens file to be sent
17         @param status Transmitter or Receiver
18 */
19 int transferFileInit(int status);
20 
21 /**
22         Closes file
23 */
24 void transferFileClose();
25 
26 /**
27         Sends file
28         @param fd Serial Port's file descriptor
29 */
30 int sendFile(int fd);
31 
32 /**
33         Receives file
34         @param fd Serial Port's file descriptor
35 */
36 int receiveFile(int fd);
```

statistics.xlsx

Ficheiro	Tamanho										
pinguim.gif	10968										
FER	**T_Prop**	**Baud Rate**	**Tamanho Tra**	**TF**	**T_Elapsed**	**A**	**P Calculada**	**P Teórica**		P = (1-FER)/(1+2*A)	
0	0	38400	428	0,15851	4,062118	0	0,5625144	1		A = T_Prop/TF	
0	50	38400	428	0,26767	6,859471	0,186794384	0,3331161	0,471366			
0	100	38400	428	0,37694	9,659462	0,265296323	0,2365556	0,308357			
0	150	38400	428	0,48621	12,459625	0,308510573	0,1833924	0,229122			
0	200	38400	428	0,59547	15,259701	0,335867453	0,1497408	0,182283			
0,25	0	38400	428	1,32913	34,060375	0	0,0670868	0,75			
0,25	50	38400	428	2,35937	60,461681	0,021192071	0,0377925	0,353524			
0,25	100	38400	428	2,21894	56,862975	0,045066523	0,0401843	0,231268			
0,25	150	38400	428	1,88325	48,260593	0,079649373	0,0473471	0,171842			
0,25	200	38400	428	1,54768	39,661131	0,129225596	0,0576131	0,136712			

Figura 9: Folha de Excel com os testes efetuados