U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

RELATÓRIO

Redes de Computadores

Trabalho Prático nº1 – Protocolo de Ligação de Dados

Turma 1

Luís Afonso Sampaio Oliveira - up201707229@fe.up.pt
Pedro Miguel Ribeiro Alves – up201707234@fe.up.pt
Ricardo França Domingues Cardoso – up201604686@fe.up.pt

Outubro de 2019

Índice

1. Sumário

Este relatório foi efetuado no âmbito da unidade curricular de Redes de Computadores, de modo a complementar o primeiro trabalho laboratorial cujo foco é a transferência de dados.

Como conclusão, todos os objetivos propostos foram cumpridos, resultando numa aplicação totalmente funcional e capaz de transferir dados sem perdas.

2. Introdução

O principal objetivo do trabalho é implementar um protocolo de ligação de dados de acordo com o guião previamente fornecido e desenvolver uma aplicação que utilize este protocolo como base para a transferência de ficheiros entre computadores através de um cabo série.

Este relatório é um suporte à aplicação desenvolvida e tem como funcionalidade explicar e analisar os vários componentes do trabalho e os seus resultados, unindo a componente prática à teórica. Isto passa pela discussão dos seguintes tópicos:

- **3. Arquitetura**

  Exposição dos blocos funcionais e interfaces.

- **4. Estrutura do Código**

  Discussão sobre a API, estruturas de dados utilizadas e funções de destaque.

- **5. Casos de uso principais**

  Identificação dos casos de uso e análise das principais sequências de chamada de funções.

- **6. Protocolo de ligação lógica**

  Identificação dos principais aspetos funcionais e descrição da estratégia de implementação destes aspetos com apresentação de extratos de código.

- **7. Protocolo de aplicação**

  Identificação dos principais aspetos funcionais e descrição da estratégia de implementação destes aspetos com apresentação de extratos de código.

- **8. Validação**

  Descrição dos testes efetuados com apresentação quantificada dos resultados.

- **9. Eficiência do protocolo de ligação de dados**

  Caraterização estatística da eficiência do protocolo.

- **10. Conclusão**

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

3. Arquitetura

O projeto encontra-se dividido em dois blocos principais, sendo estes o bloco do emissor, responsável pelo envio do ficheiro, e o bloco do recetor, responsável pelo envio do mesmo.

Existe também uma divisão entre a camada do protocolo de transmissão de dados e a camada da aplicação, que é comum a todos os elementos do trabalho.

4. Estrutura do Código

O código está distribuído por cinco ficheiros de código. Os ficheiros **"llfunctions.c"** e **"msg_state_machine.c"** tratam do Protocolo de Ligação de Dados, contendo o primeiro as funções especificadas no guião e as funções necessárias à sua implementação, e o segundo a definição da máquina de estados utilizada na leitura de tramas. O ficheiro **"app_package_handling.c"** contém as funções relativas ao Protocolo da Aplicação de Teste. O ficheiro "**noncanonical.c**" é responsável pelas funções do recetor e "**writenoncanonical.c**" é responsável pelas funções do emissor, e ambos os ficheiros recorrem às funções do Protocolo de Ligação de Dados e do Protocolo da Aplicação de Teste. Existe também, para cada ficheiro, um *header file*, onde estão declaradas as funções necessárias e constantes importantes.

llfunctions

Funções principais da camada de ligação:

- **llopen** – envia a trama de supervisão SET e recebe a trama UA, se for chamada pelo emissor; recebe a trama de supervisão SET e envia a trama UA, se for chamada pelo recetor;
- **llwrite** – faz stuffing das tramas I e envia-as;
- **llread** – recebe tramas I e recupera as tramas originais realizando destuffing;
- **llclose –** envia a trama de supervisão DISC, recebe DISC e envia UA, se for chamada pelo emissor; recebe a trama de controlo DISC, envia DISC e recebe UA, se for chamada pelo recetor.
- **serialReadControl2 –** lê o serial port caráter a caráter até a sequência desejada ser encontrada ou um alarme ser accionado. Utilizado para determinar a receção de tramas.

Variáveis Globais:

- **numAlarms** – contador do número de alarmes realizados, inicializado a 0;
- **alarmFlag** – inicializada a FALSE, passa a TRUE sempre que um alarme é acionado;
- **oldtio, newtio –** structs termios com as definições da porta série;
- **curr_ns –** número sequencial da trama a enviar, inicializado a 0.

Macros Principais:

- **MAX_RETR –** Número máximo de tentativas de reenvio;
- **TIMEOUT –** Número de segundos de cada alarme;
- **BAUDRATE –** Capacidade da Ligação;
- **D_MAX_SIZE –** Número máximo de bytes das tramas I;

msg_state_machine

Função da máquina de estados:

stateMachineOpen – altera o estado da máquina de estados conforme o caráter recebido e o estado atual. Utilizada para ler tramas de controlo e o header das tramas I.

Estados:

- **START_S –** Aguardando uma flag delimitadora que dê início à trama;
- **FLAG_RCV –** Recebeu a flag com sucesso;
- **A_RCV –** Recebeu o Campo de Endereço com sucesso;
- **C_RCV –** Recebeu o Campo de Controlo com sucesso;
- **BCC_OK –** Recebeu o BCC com sucesso;
- **STOP_S –** Terminou o processamento com sucesso.

Macros Principais:

- **F_FLAG –** flag delimitadora das tramas;
- **A1_ -** Campo de endereço em Comandos enviados pelo Emissor e Respostas enviadas pelo Receptor;
- **A2_ -** Campo de endereço em Comandos enviados pelo Receptor e Respostas enviadas pelo Emissor;
- **C_SET –** Campo de controlo do Comando SET;
- **C_I(n) –** Campo de controlo da trama I, com n = Ns;
- **C_DISC –** Campo de controlo do Comando DISC;
- **C_UA –** Campo de controlo da Resposta UA;
- **C_RR(n) –** Campo de controlo da Resposta RR, com n = Nr;
- **C_REJ(n) –** Campo de controlo da Resposta REJ, com n = Nr;
- **ESC_OCT –** Octeto de Escape, utilizado no stuffing da trama I;
- **FLAG_STUFF –** Octeto utilizado no stuffing da ocorrência da Flag nos dados ou no bcc dos dados na trama I;
- **ESC_STUFF –** Octeto utilizado no stuffing da ocorrência do Octeto de Escape nos dados ou no bcc dos dados na trama I;

app_package_handling

Funções principais da camada da aplicação:

- **buildControlPackage –** Constrói um pacote de controlo;
- **buildDataPackage –** Constrói um pacote de dados;
- **readStartPackage –** Lê o pacote de controlo *start* e guarda a informação recebida.
- **readPackage –** Lê um pacote e determina se se trata de um pacote de dados ou do pacote de controlo *endI,* processando a informação recebida se for de dados.

- **stateMachineApp –** altera o estado da máquina de estados conforme o estado atual. Utilizada para identificar o estado da aplicação;
- **displayCompletion –** mostra o progresso da transferência.

Macros Principais:

- **C_DATA –** Campo de Controlo do pacote de dados;
- **C_START –** Campo de Controlo do pacote *start;*
- **C_END –** Campo de Controlo do pacote *end;*
- **T_SIZE –** campo de *type* do tamanho do ficheiro em formato *tlv*;
- **T_NAME –** Campo de *type* do nome do ficheiro em formato *tlv*;
- **PACKAGE_DATA_SIZE –** número máximo de bytes que podem ser enviados em cada pacote de dados;

noncanonical

Funções principais da camada da aplicação:

- **isFileSizeExpected –** compara o tamanho do ficheiro recebido e o tamanho do ficheiro original;
- **createReceivedFile –** verifica se já existe um ficheiro com o mesmo nome e cria o ficheiro;
- **main –** base da camada da aplicação, visto que gere todo o processo nesta camada e chama as funções da camada de ligação.

writenoncanonical

Funções principais da camada da aplicação:

- **sendMessage –** envia o pacote pelo serial port, chamando **llwrite**, e determina se é necessário reenviar;
- **fileDataReading –** lê dados do ficheiro e constrói um pacote de dados. Muda o estado da aplicação se chegar ao fim do ficheiro.
- **setupControlPackage –** determina o tamanho do ficheiro, constrói o pacote de controlo e muda o estado da aplicação.
- **main –** base da camada da aplicação, visto que gere todo o processo nesta camada e chama as funções da camada de ligação.

5. Casos de Uso Principais

Os casos de uso da aplicação são a interface, que permite que o transmissor escolha o ficheiro a enviar, e a transferência do ficheiro por porta série, entre dois computadores, o transmissor e o recetor.

Interface

O utilizador, de modo a iniciar a aplicação, deve inserir um conjunto de argumentos. No caso do emissor o utilizador deve inserir a porta de série (ex: **/dev/ttyS0**), e o nome do ficheiro a transmitir (ex: **pinguim.gif**). No caso do recetor o utilizador necessita apenas de introduzir a porta série.

Sequência da transmissão de dados

I. Transmissor escolhe o ficheiro a enviar;
II. Configuração da ligação entre os computadores;
III. Estabelecimento da ligação;
IV. Envio da informação por parte do emissor;
V. Receção da informação por parte do recetor;
VI. Armazenamento da informação recebida pelo recetor num ficheiro com um nome igual, se não existir um ficheiro com igual nome, ou com um sufixo adequado, se já existir;
VII. Terminação da ligação.

6. Protocolo de Ligação Lógica

As escritas no serial port são feitas trama a trama, no entanto a leitura é feita caráter a caráter.

llopen

```
int llopen(char *port, int type);
```

Esta função estabelece a ligação entre o emissor e o recetor. Como tal, a abertura da porta de série e respetiva configuração é também realizada no início desta função, de modo a evitar a repetição de código.

No emissor, esta função envia a trama de controlo SET e ativa um alarme, que é desativado após receber uma resposta (UA). Se não receber uma resposta dentro de um determinado intervalo de tempo, SET é reenviado. Este mecanismo de retransmissão é repetido até um número máximo de vezes, terminando o programa se esse número for atingido sem obter nenhuma resposta.

No recetor, a função aguarda a chegada de uma trama de controlo SET, enviando uma resposta UA após a sua receção. Para enviar tanto UA como SET é utilizada a função **sendControlMessage.** Esta função tem como argumentos o Campo de Controlo e o Campo de Endereço das tramas de Supervisão/Não Numeradas, permitindo o envio de qualquer trama destes tipos.

Na receção de UA e SET é utilizada a função **serialReadControl1**, versão mais simples de **serialReadControl2**, que utiliza uma máquina de estados de modo a registar mais facilmente a ocorrência da trama desejada. A função termina quando se encontra a sequência desejada ou com a ocorrência de um alarme, o que permite interromper a leitura do serial port para reenviar tramas. A função recebe como argumento um Campo de Controlo e um Campo de Endereço, permitindo ler ambas as tramas de supervisão utilizadas em **llopen**.

llwrite

```
int llwrite(int fd, uint8_t *buffer, int length);
```

Esta função é responsável pelo stuffing e envio das tramas realizados pelo emissor.

É utilizado um número de sequência, **Ns** no emissor, **Nr** no emissor, para sincronizar o emissor com o recetor.

Primeiro é realizado o *framing* da mensagem, isto é, acrescenta-se o cabeçalho do Protocolo de Ligação à mensagem (o cálculo do BCC2 é realizado pela função **dataBCC).** É realizado o *stuffing*, quer da mensagem quer do BCC2, recorrendo à função **byteStuffing**. Após o stuffing a mensagem é enviada.

O envio da trama I possui o mesmo mecanismo de retransmissão que o envio de SET utilizado em **llopen.** Após o envio da trama é accionado um alarme que é desativado após a receção de uma resposta RR ou REJ. Caso o alarme seja atingido,

a trama é reenviada até um número máximo de vezes. Se receber um REJ é registada a necessidade de reenviar a mensagem. A verificação do tipo de resposta é realizada através da função **serialReadControl2,** que, tendo como argumentos o Campo de Endereço e dois Campos de Controlo (C de REJ e C de RR), utiliza uma máquina de estados até atingir o estado final, ao encontrar a flag de delimitadora que determina o fim da trama, ou um alarme ser accionado.

Ns apenas é alterado após a receção de uma resposta RR com **Nr** diferente, permanecendo igual no caso de reenvio, time-out e ocorrência de alarm.

llread

```
int llread(int fd, uint8_t *buffer);
```

Esta função é responsável pela receção e destuffing das tramas por realizadas pelo emissor.

O *header* é validado pela função **serialReadControl2,** que recebe como argumento um Campo de Endereço e dois possíveis Campos de Controlo (**C** quando **Ns** é 1 e **C** quando **Ns** é 0). Após receber um *header* válido, a função **readIFrame** processa o resto da mensagem, realizando o destuffing e verificando o BCC2. Caso o Campo de Controlo corresponda a um **Ns** igual ao **Nr** atual, significa que a informação já foi recebida e se trata de um duplicado, que é descartado, e envia-se uma resposta RR. Caso contrário, se houver erros nos dados, a trama também é descartada, enviando uma resposta REJ. Se não houver erros nos dados e o número de sequência for o esperado, a mensagem é processada.

De modo a permitir o teste da eficiência com a variação de FER, é chamada a função **hasError,** utilizada para simular a ocorrência de erros no BCC1 e no BCC2, com probabilidades constantes e independentes, definidas antes de o programa correr.

llclose

```
int llclose(int fd, int type);
```

Esta função trata da terminação entre o emissor e o recetor, tal como da restauração da configuração da porta de série e o fecho desta.

No emissor, é enviada a trama de Supervisão DISC com utilizando a função **sendControlMessage** e aguardada a receção de outro DISC pela função **serialReadControl1**, sendo utilizado o sistema de retransmissão previamente mencionado. Após a receção do outro DISC é enviado um UA recorrendo, novamente, a **sendControlMessage**.

No recetor é aguardado um DISC, recorrendo a **serialReadControl1**. Após a receção deste DISC envia-se um DISC, utilizando a função **sendControlMessage,** e espera-se um UA, utilizando a função **serialReadControl1**, com o uso do sistema de retransmissão.

7. Protocolo de Aplicação

O protocolo de aplicação implementado está implementado da seguinte forma:

- O envio dos pacotes START e END indicam, respetivamente, o início e fim da transferência do ficheiro. Ambos contém o nome e tamanho do ficheiro enviado.
- O ficheiro é lido progressivamente pelo emissor, conforme vai enviando os dados, lendo o máximo possível de bytes a cada iteração. O número de bytes lidos não pode ultrapassar o número máximo atribuído ao campo de dados dos pacotes de dados
- Cada fragmento de dados é encapsulado com um header que contém o número de sequência, em módulo 255, do pacote e o tamanho do fragmento.
- O ficheiro é construído progressivamente pelo recetor, conforme vai recebendo os fragmentos de dados.

buildControlPackage

```
unsigned int buildControlPackage(uint8_t control_field, unsigned int fi
le_size, uint8_t *file_name, int file_name_length, uint8_t *dest);
```

Esta função cria um pacote START ou END, colocando-o no apontador *dest*. Recebe como argumentos o campo de controlo, para identificar o pacote pretendido, o nome do ficheiro e o tamanho deste. Este pacote será enviado pela função **llwrite**, pertencente ao protocolo de ligação.

buildDataPackage

```
unsigned int buildDataPackage(unsigned int data_size, uint8_t *data, ui
nt8_t *dest);
```

Esta função cria um pacote de dados. Recebe como argumentos os dados, o seu tamanho e um apontador para armazenar o pacote construído. O header é então produzido, sendo constituído por:

- Campo de Controlo C, que identifica o pacote como sendo de dados;
- Número de Sequência N, em módulo 255;
- Os octetos L2 e L1, que representam o tamanho K do ficheiro (K = L2 * 256 + L1);

createReceivedFile

```
int createReceivedFile(char *file_name)
```

Esta função verifica se o ficheiro a ser recebido pelo recetor já existe. Caso exista, tenta, sequencialmente, nomes alternativos até alcançar um que não esteja a ser utilizado. As alternativas são construídas acrescentando um sufixo ao nome do ficheiro (ex: pinguim(1).gif). A função também distingue nomes que contenham extensões ou não, de modo a colocar o sufixo de forma adequada.

sendMessage

```
int sendMessage(int bytes_written, int fd, uint8_t *data_package);
```

Esta função envia o pacote já criado pelo serial port, retornando TRUE se for necessário reenviá-lo, FALSE se não for, e terminando o programa se ocorrer time-out.

fileDataReading

```
int fileDataReading(int fd, uint8_t *data_package, enum appState *state
)
```

Esta função lê bytes do ficheiro até atingir o número máximo permitido por pacote ou até chegar ao fim do ficheiro. Se não houver mais bytes para ler, altera o estado da aplicação, senão constrói um pacote com os dados recebidos.

8. Validação

De modo a testar a aplicação desenvolvida, foram realizados os seguintes testes:

- Envio de ficheiros de diversos tamanhos;
- Geração de curto circuito durante o envio de um ficheiro;
- Interrupção da ligação por alguns segundos, aquando do envio do ficheiro;
- Envio de um ficheiro com variação na percentagem de erros simulados;
- Envio de um ficheiro com variação do tamanho do pacote;
- Envio de um ficheiro com variação das capacidades de ligação (Baudrate);

Todos os testes foram concluídos com sucesso.

9. Eficiência do Protocolo de Ligação de Dados

Variação da Capacidade de Ligação

De acordo com os testes realizados, a eficiência do protocolo diminui com o aumento do baudrate.

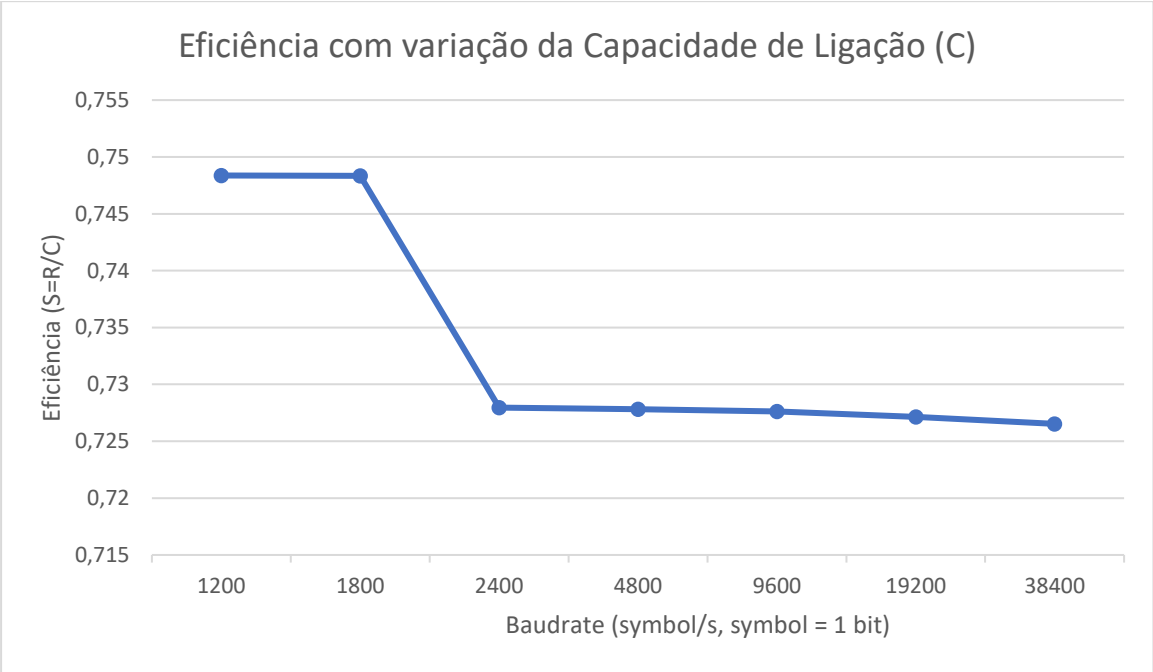

Gráfico 1 - Eficiência com variação do baudrate

Variação do Tempo de Propagação (T_prop)

O aumento do tempo de propagação leva a uma diminuição da eficiência. O transporte de cada pacote provoca um atraso, que, acumulado, não é negligível, como comprova o gráfico.

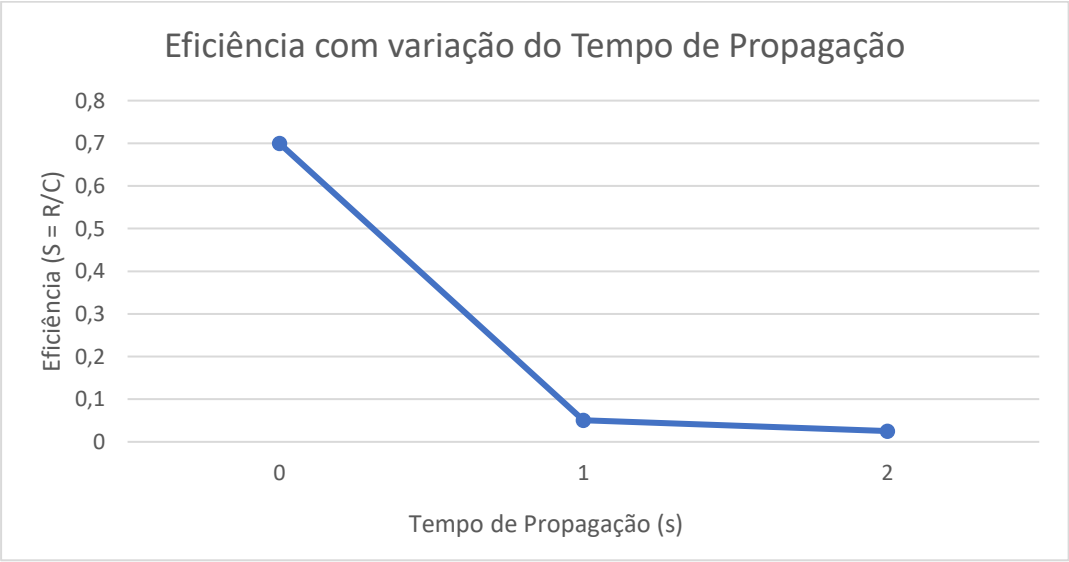

Gráfico 2 - Eficiência com variação do tempo de propagação

Variação do tamanho dos pacotes de dados

O gráfico abaixo demonstra que do aumento do tamanho dos pacotes de dados resulta o aumento da eficiência. Um maior número de dados transportados significa um menor número de tramas enviadas, o que faz com que o programa execute mais rapidamente.

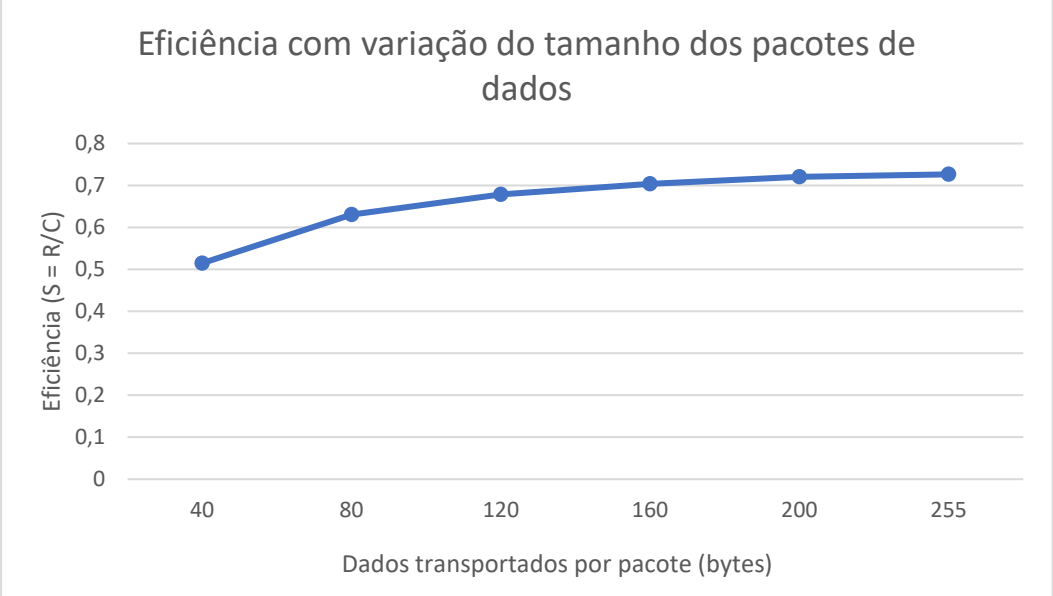

Gráfico 3 - Eficiência com variação do tamanho dos pacotes de dados

Variação de FER

O gráfico comprova que a ocorrência de erros no BCC1 e no BCC2 afetam negativamente a eficiência do programa. No caso de erros no BCC1, o efeito é severo, pois o recetor não responde, levando o emissor a aguardar um número previamente definido de segundos antes de reenviar a informação. Isto atrasa bastante a execução, e se a probabilidade de erros no BCC1 for alta, há a possibilidade de o recetor receber vários pacotes inválidos seguidos, o que pode provocar um time-out no emissor. No caso de erros no BCC2, o atraso é menor, visto que o programa reenvia imediatamente a trama.

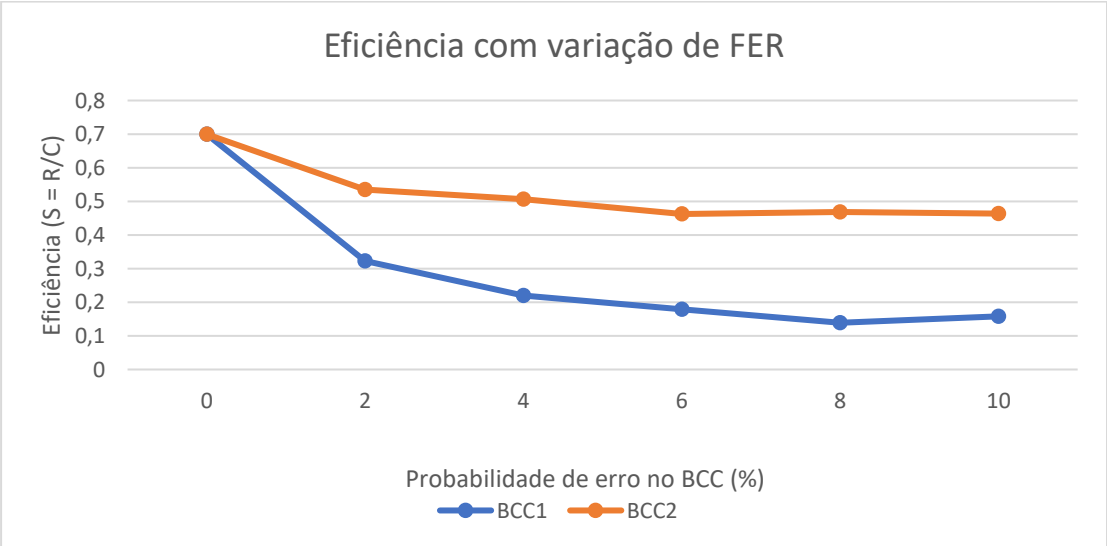

Gráfico 4 - Eficiência com variação de FER

10. Conclusões

O tema do trabalho é o protocolo de ligação de dados e consiste em implementar um sistema de comunicação de dados entre dois sistemas ligados por um *serial port*. Este sistema deve ser fiável, garantindo o envio de dados sem erros.

Também foi desenvolvida uma aplicação, de modo a testar o protocolo realizado, sendo independente deste. Esta independência foi alcançada utilizando funções especiais, que serviram de interface protocolo-aplicação. Assim, na camada de ligação de dados não há qualquer processamento dos pacotes de dados a serem transportados na camada da aplicação, e na camada da aplicação não são conhecidos os detalhes da implementação do protocolo de ligação de dados. Apenas é conhecida a forma de acesso ao sistema implementado.

Em conclusão, o trabalho foi concluído com sucesso, cumprindo todos os objetivos, e permitiu aprofundar os conhecimentos teóricos e práticos dos temas abordados.

11. Anexo I - Código Fonte

llfunctions.h

```
#ifndef LLFUNCTIONS_H
#define LLFUNCTIONS_H

#include "msg_state_machine.h"
#include <termios.h>

#define BIT(n) (0x01 << (n))

#define TRANSMITTER 0
#define RECEIVER 1

#define COMMAND 0
#define REPLY 1

#define FALSE 0
#define TRUE 1

#define MAX_RETR 3
#define TIMEOUT 3
#define BAUDRATE B38400

//ll return values
#define TIMEOUT_RET -1
#define RESEND_RET -2

//I FRAME data max size
```

```
#define D_MAX_SIZE         255
#define D_STUFFED_MAX_SIZE D_MAX_SIZE * 2

//I frame max size
#define HEAD_SIZE               4
#define F_TAIL_SIZE             2
#define F_STUFFED_TAIL_SIZE     3
#define F_STUFFED_SIZE          HEAD_SIZE + D_STUFFED_MAX_SIZE + F_STUFFED_TA
IL_SIZE

//I frame return values
#define DATA_ERROR -1

//reply types
#define RR  1
#define REJ 2

//discard frame
#define DISCARD -1



/**
 * @brief Indicates the alarm call through a global flag and increments the al
arm counter
 * @param sig signal identifier.
 */
void alarmHandler(int sig);

/**
 * @brief Assembles the control frame and sends it to the given file descripto
r
 *
 * @param fd     The file descriptor.
 * @param C      Frame control field.
 * @param A      Frame adress field.
 */
void sendControlMessage(int fd, uint8_t C, uint8_t A);

/**
 * @brief Assembles the I frame and sends it to the given file descriptor. Han
dles stuffing, head and tail building.
 *
 * @param fd     The file descriptor.
 * @param c_num  I Frame control field. Can either be C_I(0) or C_I(1).
 * @param info   Information to be transferred, not yet stuffed.
 * @param info_length   Length of information (number of bytes/octets)
 * @return int   Number of bytes written (after stuffing).
 */
```

```c
int sendICommand(int fd, int c_num, uint8_t *info, unsigned int info_length);

/**
 * @brief Reads the I frame from the file descriptor and retrieves the message
 it carries. Handles destuffing, tail reception and validation.
 *
 * @param fd     The file descriptor.
 * @param buffer Where the message will be stored (unstuffed).
 * @return int   Number of bytes read (after destuffing), or DATA_ERROR if the
 data is not valid.
 */
int readIFrame(int fd, uint8_t *buffer);


/**
 * @brief Assembles I frame's Data BCC, obtained through the exclusive OR of a
ll the data's octets.
 *
 * @param info         The Data.
 * @param info_length  Data's length, in bytes.
 * @return uint8_t     Octet obtained through the XOR of all data's octets
 */
uint8_t dataBCC(uint8_t *info, unsigned int info_length);

/**
 * @brief Receives a message and builds the corresponding stuffed message
 *
 * @param msg          The Data.
 * @param msg_length   Data's length, in bytes.
 * @param stuffed_msg Where the stuffed message will be stored.
 * @return int         Size of stuffed message.
 */
int byteStuffing(uint8_t *msg, unsigned int msg_length, uint8_t *stuffed_msg);

/**
 * @brief Receives a stuffed message and rebuilds the original message
 *
 * @param stuffed_msg Stuffed message.
 * @param msg_length  Data's length, in bytes.
 * @param destuff     Where the original message will be stored
 * @return int        Size of original message.
 */
int destuffing(uint8_t *stuffed_msg, unsigned int msg_length, uint8_t *destuff
);


/**
 * @brief Reads from the file descriptor until it receives the ending flag or
exceeds the message max size. Handles the
```

```
 * retrieval of the I frame's half after the tail (data + head) for ulterior p
rocessing.
 *
 * @param fd          The file descriptor.
 * @param buffer      Where the received data will be stored.
 * @return int        Number of bytes read, a negative value if it fails or ex
ceeds the max size.
 */
int getFrame(int fd, uint8_t *buffer);

/**
 * @brief Compares the received Data BCC with a BCC calculated with the receiv
ed data to check if there were transmission errors.
 *
 * @param msg     The Data.
 * @param length  Data's length, in bytes.
 * @param BCC     BCC read from the serial port, used in error checking.
 * @return int    0 if the BCCs differ (due to a transmission error), other va
lue if they're the same.
 */
int checkBCC(uint8_t *msg, unsigned length, uint8_t BCC);

/**
 * @brief Opens the serial port and sets the signal handler, then attempts to
establish the connection.
 * a) type == TRANSMITTER
 *
 * Sends the SET and waits for the UA reply. When SET is sent, an alarm is cal
led, and if the reply is not received
 * within TIMEOUT seconds it resends the SET command. It keeps re-
sending until it receives the reply or has attempted
 * MAX_RETR times.
 *
 * b) type == RECEIVER
 * Awaits the SET command and sends the UA reply when it receives it.
 *
 * @param port    The Port to be opened.
 * @param type    App type indentifier (TRANSMITTER or RECEIVER).
 * @return int    Returns a file descriptor if it sucessfully establishes the
connection, 0 if there's an error when acessing the serial
 * port.
 * a) returns TIMEOUT_RET if it times out while sending the SET command
 */
int llopen(char *port, int type);

/**
 * @brief Sends a message through the serial port. Uses Ns to synchronize with
 the receiver.
```

```
 * Sends the I frame and waits for the reply. When SET is sent, an alarm is ca
lled, and if the reply is not received
 * within TIMEOUT seconds it resends the SET command. It keeps re-
sending until it receives the reply or has attempted
 * MAX_RETR times.
 * @param fd      File descriptor.
 * @param buffer  Message to be sent through the serial port.
 * @param length  Message's size.
 * @return int    Returns the number of bytes written on a sucessful transmitt
ion, RESEND_RET if the same data has to be sent again
 * or TIMEOUT_RET if it times out.
 */
int llwrite(int fd, uint8_t *buffer, int length);

/**
 * @brief Receives a message from the serial port. Uses Nr to synchronize with
 the transmitter.
 * Receives the I frame. If the data is ok and carries the expected Ns then re
plies with an RR reply;
 * if the data is a duplicate (the Ns is different than expected) it also send
s the RR reply.
 * If the data has errors it sends the REJ reply.
 * @param fd      File descriptor.
 * @param buffer  Where the received message will be stored.
 * @param length  Message's size.
 * @return int    Returns the number of bytes written on a sucessful transmitt
ion,
 * DISCARD if the data has errors or is a duplicate.
 */
int llread(int fd, uint8_t *buffer);

/**
 * @brief Attempts to terminate the connection.
 * a) type == TRANSMITTER
 *
 * Sends the DISC command and waits for the DISC reply. When DISC is sent, an
alarm is called, and if the reply is not received
 * within TIMEOUT seconds it resends the DISC command. It keeps re-
sending until it receives the reply or has attempted
 * MAX_RETR times. If it receives the DISC reply, it sends an UA reply to the
receiver.
 *
 * b) type == RECEIVER
 * Awaits the DISC command and sends the DISC reply when it receives it.
 * It then waits for the UA reply. When DISC is sent, an alarm is called, and
if the reply is not received
 * within TIMEOUT seconds it resends DISC. It keeps re-
sending until it receives the reply or has attempted
```

```
 * MAX_RETR times.
 * @param port     The Port to be opened.
 * @param type     App type indentifier (TRANSMITTER or RECEIVER).
 * @return int     Returns 1 if sucessful, TIMEOUT_RET if it times out.
 */
int llclose(int fd, int type);

/**
 * @brief Reads the serial until either a control frame or I frame's header wi
th the specified A and C fields has been found.
 * It uses a state machine to process the received bytes.
 * It can search for two C's at the same time, testing each one in a different
 state machine.
 * @param fd       File descriptor.
 * @param c        Desired control field.
 * @param a        Desired adress field.
 * @param c2       Another desired control field, used when there are two possi
ble values for C.
 * @param headerI Determines if it's looking for a control frame or an I frame
's header. If TRUE, it stops at the BCC,
 * otherwise looks for the end flag.
 * @return int     Returns 1 if it found the desired sequence, while using c; 2
 if it did so while using c2;
 * FALSE if it didn't find anything when the alarm was called, -
1 if there was an error reading.
 */
int serialReadControl2(int fd, uint8_t c, uint8_t a, uint8_t c2, int headerI);

/**
 * @brief serialReadControl version which can only search for a control frame
and test a single C octet.
 */
int serialReadControl1(int fd, uint8_t c, uint8_t a);

/**
 * @brief initializes the port
 * */
void initPort(int fd);

#endif
```

llfunctions.c

```c
#include "llfunctions.h"
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <termios.h>
#include <stdint.h>
#include <signal.h>
#include <errno.h>

int numAlarms = 0;
int alarmFlag = FALSE;
struct termios oldtio, newtio;
int curr_ns = 0;

void alarmHandler(int sig)
{
    if (sig == SIGALRM)
    {
        numAlarms++;
        alarmFlag = TRUE;
        printf("alarm nº %d called\n", numAlarms);
    }
}

void sendControlMessage(int fd, uint8_t C, uint8_t A)
{
    uint8_t msg[5];
    msg[0] = F_FLAG;
    msg[1] = A;
    msg[2] = C;
    msg[3] = A ^ C;
    msg[4] = F_FLAG;
    write(fd, msg, 5);
}

int sendICommand(int fd, int c_num, uint8_t *info, unsigned int info_length)
{
    uint8_t msg[F_STUFFED_SIZE];
    uint8_t unstuffed_bcc;
    msg[0] = F_FLAG;
```

```
    msg[1] = A1_;
    msg[2] = C_I(c_num);   //C
    msg[3] = A1_ ^ msg[2]; //BCC 1

    // for (unsigned int i = 0; i < info_length; i++)
    // {
    //     printf("%x", info[i]);
    // }
    // printf("\n");

    unsigned int header_size = 4;

    unsigned int stuffed_msg_size = byteStuffing(info, info_length, msg + head
er_size); //stuffs the message

    unstuffed_bcc = dataBCC(info, info_length); //BCC 2, XOR of all the data o
ctets

    unsigned int stuffed_bcc_size = byteStuffing(&unstuffed_bcc, 1, msg + head
er_size + stuffed_msg_size); //stuffs the bcc

    msg[stuffed_msg_size + header_size + stuffed_bcc_size] = F_FLAG;

    unsigned int tail_size = stuffed_bcc_size + 1;

    unsigned int frame_size = header_size + stuffed_msg_size + tail_size;

    // for (unsigned int i = 0; i < frame_size; i++)
    // {
    //     printf("%x", msg[i]);
    // }
    // printf("\n");

    return write(fd, msg, frame_size);
}

int readIFrame(int fd, uint8_t *buffer)
{

    uint8_t stuffed_msg_w_tail[D_STUFFED_MAX_SIZE + F_STUFFED_TAIL_SIZE];
    uint8_t unstuffed_msg_w_tail[D_MAX_SIZE + F_TAIL_SIZE];
    int stuffed_msg_size = getFrame(fd, stuffed_msg_w_tail);

    // printf("STUFFED\n");
    // for (unsigned int i = 0; i < stuffed_msg_size; i++)
    // {
    //     printf("%x", stuffed_msg_w_tail[i]);
    // }
```

```
    // printf("\n");
    if (stuffed_msg_size <= 0)
    {
        return DATA_ERROR;
    }

    int unstuffed_msg_size = destuffing(stuffed_msg_w_tail, stuffed_msg_size,
unstuffed_msg_w_tail) - F_TAIL_SIZE; //destuffs message

    if (unstuffed_msg_size <= 0)
    {
        return DATA_ERROR;
    }

    uint8_t bcc = unstuffed_msg_w_tail[unstuffed_msg_size]; //retrieves BCC fr
om tail

    // printf("UNSTUFFED\n");
    // for (unsigned int i = 0; i < unstuffed_msg_size + F_TAIL_SIZE; i++)
    // {
    //     printf("%x", unstuffed_msg_w_tail[i]);
    // }
    // printf("\n");

    int is_bcc_valid = checkBCC(unstuffed_msg_w_tail, unstuffed_msg_size, bcc)
; //compares the received BCC and the BCC calculated with the received data
    //printf("is bcc valid? %d\n", is_bcc_valid);

    //error simulation

    if (hasError(BCC2ERR))
        is_bcc_valid = 0;

    if (is_bcc_valid)
    {
        memcpy(buffer, unstuffed_msg_w_tail, unstuffed_msg_size);
        return unstuffed_msg_size;
    }
    else
    {
        return DATA_ERROR;
    }
}

uint8_t dataBCC(uint8_t *info, unsigned int info_length)
{

    uint8_t bcc = info[0];
```

```
    for (unsigned i = 1; i < info_length; i++)
    {
        bcc = bcc ^ info[i];
    }
    return bcc;
}

//builds stuffed msg, returns size of msg after stuffing
int byteStuffing(uint8_t *msg, unsigned int msg_length, uint8_t *stuffed_msg)
{

    unsigned j = 0;
    for (unsigned i = 0; i < msg_length; i++)
    {
        uint8_t curr_char = msg[i];
        //printf("%x\n", curr_char);
        if (curr_char == F_FLAG)
        {
            stuffed_msg[j++] = ESC_OCT;
            stuffed_msg[j++] = FLAG_STUFF;
        }
        else if (curr_char == ESC_OCT)
        {
            stuffed_msg[j++] = ESC_OCT;
            stuffed_msg[j++] = ESC_STUFF;
        }
        else
        {
            stuffed_msg[j++] = curr_char;
        }
        //printf("%d - size\n", j);
    }
    return j;
}

//builds a destuffed message. returns size of destuffed msg
int destuffing(uint8_t *stuffed_msg, unsigned int msg_length, uint8_t *destuff
ed_msg)
{
    unsigned j = 0;
    // printf("DESTUFFING\n");
    for (unsigned i = 0; i < msg_length; i++)
    {
        uint8_t curr_char = stuffed_msg[i];
        // printf("%x", curr_char);
        if (curr_char == ESC_OCT)
        {
            i++;
```

```
            uint8_t next_char = stuffed_msg[i];
            // printf("%x", next_char);
            if (next_char == FLAG_STUFF)
            {
                destuffed_msg[j] = F_FLAG;
            }
            else if (next_char == ESC_STUFF)
            {
                destuffed_msg[j] = ESC_OCT;
            }
        }
        else
        {
            destuffed_msg[j] = curr_char;
        }
        j++;
    }
    // printf("\n");
    // printf("DESTUFFING ENDED\n");
    return j;
}

int getFrame(int fd, uint8_t *buffer)
{
    int res = 0;
    uint8_t curr_char;
    while (!alarmFlag)
    { /* loop for input */
        if (res >= D_STUFFED_MAX_SIZE + F_STUFFED_TAIL_SIZE)
            return -2;
        if (read(fd, &curr_char, 1))
        {
            buffer[res] = curr_char;
            // printf("%x", buffer[res]);
            res++;
        }
        else
        {
            return -1;
        }
        if (curr_char == F_FLAG && res > 0)
            break;
    }

    // printf("\n");
    // printf("RES - %d\n\n", res);
    return res;
}
```

```
int checkBCC(uint8_t *msg, unsigned length, uint8_t bcc)
{
    uint8_t data_bcc = msg[0];
    for (unsigned i = 1; i < length; i++)
    {
        data_bcc = data_bcc ^ msg[i];
    }
    return (data_bcc == bcc);
}

int llopen(char *port, int type)
{

    int fd;
    fd = open(port, O_RDWR | O_NOCTTY);
    if (fd < 0)
        return 0;

    initPort(fd);

    signal(SIGALRM, alarmHandler);
    siginterrupt(SIGALRM, 1);

    if (type == TRANSMITTER)
    {
        //sprintf(buf, "%lx", frame);

        do
        {
            alarmFlag = FALSE;
            printf("Sending SET command - attempt nº %d\n", numAlarms + 1);
            sendControlMessage(fd, C_SET, A1_);
            alarm(TIMEOUT);
            if (serialReadControl1(fd, C_UA, A1_) > 0)
            {
                alarm(0);
                printf("Received UA reply\n");
                break;
            }
        } while (alarmFlag && numAlarms < MAX_RETR);
    }
    else
    {
        if (serialReadControl1(fd, C_SET, A1_) > 0)
        {
            printf("Received SET command\n");
            sendControlMessage(fd, C_UA, A1_);
```

```
            printf("Sent UA reply\n");
        }
    }

    alarmFlag = FALSE;

    if (numAlarms == MAX_RETR)
    {
        numAlarms = 0;
        return TIMEOUT_RET;
    }
    else
    {
        numAlarms = 0;
        return fd;
    }
}

int llwrite(int fd, uint8_t *buffer, int length)
{
    //used to synch the transmitter and receiver
    int curr_nr = (curr_ns + 1) % 2;
    unsigned bytes_written = 0;
    int reply_type = 0;

    printf("%d - CURR NS\n", curr_ns);

    do
    {
        alarmFlag = FALSE;
        printf("Sending I frame - attempt nº %d\n", numAlarms + 1);
        bytes_written = sendICommand(fd, curr_ns, buffer, length);
        alarm(TIMEOUT);
        reply_type = serialReadControl2(fd, C_RR(curr_nr), A1_, C_REJ(curr_nr)
, FALSE);
        if (reply_type > 0)
        {
            //Received a valid reply
            alarm(0);
            break;
        }
    } while (alarmFlag && numAlarms < MAX_RETR);

    alarmFlag = FALSE;
    numAlarms = 0;
    if (reply_type == RR)
    {
        //The I frame was sucessfully received
```

```
        printf("Received RR reply\n");
        curr_ns = (curr_ns + 1) % 2;
        return bytes_written;
    }
    else if (reply_type == REJ)
    {
        //The I frame was either received with errors or a duplicate
        printf("Received REJ reply\n");
        return RESEND_RET;
    }
    else
    {
        printf("Timed out while sending I frame\n");
        return TIMEOUT_RET;
    }
}

int llread(int fd, uint8_t *buffer)
{
    int curr_nr = (curr_ns + 1) % 2;

    printf("%d - CURR NS\n", curr_ns);

    int command_type = serialReadControl2(fd, C_I(curr_ns), A1_, C_I(curr_nr),
 TRUE);
    printf("COMMAND_TYPE HEADER RESULT = %d\n", command_type);

    int msg_size = DISCARD;
    // sleep(1);
    if (command_type > 0)
    {
        msg_size = readIFrame(fd, buffer);
        printf("msg size = %d\n", msg_size);
        if (command_type == 1)
        {
            if (msg_size >= 0)
            {
                printf("Data is ok, sent RR reply\n");
                sendControlMessage(fd, C_RR(curr_nr), A1_);
                curr_ns = (curr_ns + 1) % 2;
            }
            else
            {
                printf("Error in data, Sent REJ reply\n");
                msg_size = DISCARD;
                sendControlMessage(fd, C_REJ(curr_nr), A1_);
            }
        }
```

```
        else if (command_type == 2)
        {
            printf("Frame is duplicate, sent RR reply\n");
            msg_size = DISCARD;

            sendControlMessage(fd, C_RR(curr_ns), A1_);
        }
    }

    return msg_size;
}

int llclose(int fd, int type)
{
    if (type == TRANSMITTER)
    {
        do
        {
            alarmFlag = FALSE;
            printf("Sending DISC command - attempt nº %d\n", numAlarms + 1);
            sendControlMessage(fd, C_DISC, A1_);
            alarm(TIMEOUT);
            if (serialReadControl1(fd, C_DISC, A2_) > 0)
            {
                alarm(0);
                printf("Received DISC command\n");
                sendControlMessage(fd, C_UA, A2_);
                printf("Sent UA reply\n");
                break;
            }
        } while (alarmFlag && numAlarms < MAX_RETR);
    }
    else
    {
        if (serialReadControl1(fd, C_DISC, A1_) > 0)
        {
            printf("Received DISC command\n");
            do
            {
                alarmFlag = FALSE;
                printf("Sending DISC command - attempt nº %d\n", numAlarms + 1
);
                sendControlMessage(fd, C_DISC, A2_);
                alarm(TIMEOUT);
                if (serialReadControl1(fd, C_UA, A2_) > 0)
                {
                    alarm(0);
                    printf("Received UA reply\n");
```

```
                    break;
                }
            } while (alarmFlag && numAlarms < MAX_RETR);
        }
    }

    alarmFlag = FALSE;

    if (tcsetattr(fd, TCSANOW, &oldtio) == -1)
    {
        perror("tcsetattr");
        exit(-1);
    }
    close(fd);

    if (numAlarms == MAX_RETR)
    {
        numAlarms = 0;
        return TIMEOUT_RET;
    }
    else
    {
        numAlarms = 0;
        return 1;
    }
}

int serialReadControl2(int fd, uint8_t c, uint8_t a, uint8_t c2, int headerI)
{
    enum state curr_state = START_S;
    enum state curr_state2 = START_S;

    uint8_t curr_byte;
    while (curr_state != STOP_S && curr_state2 != STOP_S && !alarmFlag)
    { /* loop for input */
        // if (headerI)
        //     printf("curr_byte = %x, curr_state1 = %d, curr_state2 = %d\n",
curr_byte, curr_state, curr_state2);
        if (read(fd, &curr_byte, 1) < 0)
        {
            if (errno != EINTR)
                return -1;
        }
        stateMachineOpen(&curr_state, &curr_byte, c, a, headerI);
        if (c2 != 0x0F)
        {
            stateMachineOpen(&curr_state2, &curr_byte, c2, a, headerI);
        }
```

```
    }
    if (curr_state == STOP_S)
        return 1;
    else if (curr_state2 == STOP_S)
        return 2;
    else
        return FALSE;
}

int serialReadControl1(int fd, uint8_t c, uint8_t a)
{
    return serialReadControl2(fd, c, a, 0x0F, FALSE);
}

void initPort(int fd)
{

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

    bzero(&newtio, sizeof(newtio));
    newtio.c_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;
    newtio.c_iflag = IGNPAR;
    newtio.c_oflag = 0;

    /* 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 5 chars received */

    /*
    VTIME e VMIN devem ser alterados de forma a proteger com um temporizador a
    leitura do(s) pr?ximo(s) caracter(es)
  */

    tcflush(fd, TCIOFLUSH);

    if (tcsetattr(fd, TCSANOW, &newtio) == -1)
    {
        perror("tcsetattr");
        exit(-1);
    }

    printf("New termios structure set\n");
```

```
}
```

msg_state_machine.h

```
/* State Machine for messages communications */

#ifndef MSG_STATE_MACHINE_H
#define MSG_STATE_MACHINE_H

#include <stdint.h>

/**/

#define F_FLAG 0x7E

/* 00000011 (0x03) em Comandos enviados pelo Emissor e Respostas enviadas pelo
 Receptor */

#define A1_ 0X03

/* 00000001 (0x01) em Comandos enviados pelo Receptor e Respostas enviadas pel
o Emissor */

#define A2_ 0X01

//comandos, n = 0/1
#define C_SET 0x03
#define C_I(n)   ((n) << (6))
#define C_DISC 0x0B

//respostas, n = 0/1
#define C_UA 0x06
#define C_RR(n)    (((n) << (7)) | 0x05)
#define C_REJ(n)   (((n) << (7)) | 0x01)


//stuffing
#define ESC_OCT     0x7D
#define FLAG_STUFF  0x5E
#define ESC_STUFF   0x5D
/**/

//
#define BCC1ERR 0
#define BCC2ERR 0
```

```
enum state {START_S, FLAG_RCV, A_RCV, C_RCV, BCC_OK, STOP_S};

void stateMachineOpen(enum state *state, uint8_t *received_char, uint8_t c, ui
nt8_t a, int headerI);

int hasError(int error_chance);


#endif //MSG_STATE_MACHINE_H
```

msg_state_machine.c

```
/* State Machine for messages communications */

#include "msg_state_machine.h"
#include <stdio.h>
#include <stdlib.h>

#define FALSE 0
#define TRUE 1

int hasError(int error_chance)
{
    int has_error = 0;
    if (error_chance > 0)
    {
        int random_number = (rand() % 100) + 1;
        printf("RANDOM NUMBER - %d\n", random_number);
        has_error = (error_chance >= random_number);
        printf("ERRORI - %d\n", has_error);
    }
    return has_error;
}

uint8_t get_bcc(uint8_t A, uint8_t C, int headerI)
{
    if (headerI)
    {
        if (hasError(BCC1ERR))
            return A & C;
    }
    return A ^ C;
```

```
}

void stateMachineOpen(enum state *state, uint8_t *received_char, uint8_t c, ui
nt8_t a, int headerI){

    switch (*state){
        //flag
        case START_S:
            if (*received_char == F_FLAG)
                *state = FLAG_RCV;
            //printf("START - RCV = %x\n", *received_char);
            break;
        case FLAG_RCV:
            if (*received_char == a)
                *state = A_RCV;
            else if (*received_char != F_FLAG)
                *state = START_S;
            //printf("FLAG - RCV = %x\n", *received_char);
            break;
        case A_RCV:
            if (*received_char == c)
                *state = C_RCV;
            else if (*received_char == F_FLAG)
                *state = 1;
            else *state = START_S;
            //printf("A - RCV = %x\n", *received_char);
            break;
        case C_RCV:
            if (*received_char == get_bcc(a, c, headerI))
            {
                if (headerI != TRUE)
                    *state = BCC_OK; //if it is a control frame, the BCC is no
t the last octet
                else
                    *state = STOP_S; //if it is an I header, the BCC is the la
st octet
            }
            else if (*received_char == F_FLAG)
                *state = FLAG_RCV;
            else
                *state = START_S;
            //printf("C - RCV = %x\n", *received_char);
            break;
        case BCC_OK:
            if (*received_char == F_FLAG)
                *state = STOP_S;
            else *state = START_S;
            //printf("BCC - RCV = %x\n", *received_char);
```

```
            break;
        default:
            break;

    }

}
```

app_package_handling.h

```
#ifndef APP_PACKAGE_HANDLING_H
#define APP_PACKAGE_HANDLING_H

#include <stdint.h>

//C
#define C_DATA     0x01
#define C_START    0x02
#define C_END      0x03

//control TLV
#define T_SIZE     0x00
#define T_NAME     0x01


#define PACK_MAX_SIZE  255
#define PACKAGE_H_SIZE 4
#define PACKAGE_DATA_SIZE PACK_MAX_SIZE - PACKAGE_H_SIZE

//end package returns
#define INVALID_END_PACKAGE -1
#define VALID_END_PACKAGE   -2
#define INVALID_PACKAGE     -3

//file
#define FILE_NAME_MAX_SIZE     255
#define EXTENSION_MAX_SIZE     16
#define ORIGINAL_FILE_MAX_SIZE PACK_MAX_SIZE - 10

//START: start of file data sending, DATA: sending file data, END: Ending Prog
ram, STOP: Program has ended.
enum appState {START, DATA, END, STOP_};

/**
 * @brief Changes the state to the next
 */
```

```
void stateMachineApp(enum appState *state);

/**
 * @brief Builds a TLV format message
 * @param type      T.
 * @param length    L.
 * @param value     V.
 * @param dest      Where the TLV message will be stored.
 * @return int      Size of tlv message (in bytes).
 */
unsigned int writeTlv(uint8_t type, uint8_t length, uint8_t *value, uint8_t *d
est);

/**
 * @brief Builds a Control Package, which is made of a Control Field and
 * two TLV messages, containing the file size and file name
 * @param dest      Where the Control Package will be stored.
 * @return int      Size of Control Package (in bytes).
 */
unsigned int buildControlPackage(uint8_t control_field, unsigned int file_size
, uint8_t *file_name, int file_name_length, uint8_t *dest);

/**
 * @brief Builds a Data Package, which is made of a Control Field, a sequence
number (N),
 * two octets which contain the data's size, (L1 and L2), and the data.
 * @param dest      Where the Data Package will be stored.
 * @return int      Size of Data Package (in bytes).
 */
unsigned int buildDataPackage(unsigned int data_size, uint8_t *data, uint8_t *
dest);

/**
 * @brief Checks if the received end_package is valid (has the same TLV data a
s the start_package)
 * @return int      TRUE if is valid, FALSE if it isn't.
 */
int isEndPackage(uint8_t *package, uint8_t *start_package, unsigned int start_
package_length);

/**
 * @brief Reads the Start Package, checking the control field and retrieving t
he values sent in TLV format
 * @param package       Received Package.
 * @param package_size  Received Package's size (in bytes).
 * @param file_size     Where the file size will be stored.
 * @param file_name     Where the file name will be stored.
```

```
 * @return int          Positive value if it's a start package, negative other
wise
 */
int readStartPackage(uint8_t *package, unsigned int package_size, uint8_t *fil
e_size, uint8_t *file_name);

int readDataPackage(uint8_t *package, uint8_t *dest, uint8_t expected_N);

int readPackage(uint8_t *package, uint8_t *dest, uint8_t expected_N, uint8_t *
start_package, unsigned int start_package_length);

void display_completion(unsigned int file_size, unsigned int curr_size);


#endif
```

app_package_handling.c

```
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include "app_package_handling.h"

#define FALSE 0
#define TRUE 1

void stateMachineApp(enum appState *state)
{

    switch (*state)
    {
    case START:
        *state = DATA;
        break;
    case DATA:
        *state = END;
        break;
    case END:
        *state = STOP_;
        break;
    default:
        break;
    }
}
```

```c
unsigned int writeTlv(uint8_t type, uint8_t length, uint8_t *value, uint8_t *dest)
{

    dest[0] = type;
    dest[1] = length;
    memcpy(dest + 2, value, length);
    return length + 2;
}

unsigned int buildControlPackage(uint8_t control_field, unsigned int file_size, uint8_t *file_name, int file_name_length, uint8_t *dest)
{

    dest[0] = control_field;

    unsigned int total_package_size = 1;

    //tlv file size
    //printf("SIZE OF FILE SIZE - %x\n", sizeof(file_size));
    total_package_size += writeTlv(T_SIZE, sizeof(file_size), (uint8_t *) &file_size, dest + total_package_size);
    //printf("FILE SIZE - %x\n", file_size);
    //tlv file name
    total_package_size += writeTlv(T_NAME, file_name_length + 1, file_name, dest + total_package_size);

    // for (int i = 0; i < total_package_size; i++){
    //     printf("%x ", dest[i]);
    // }
    // printf("\n");

    return total_package_size;
}

unsigned int buildDataPackage(unsigned int data_size, uint8_t *data, uint8_t *dest)
{
    static uint8_t sequence_number_N = 0;

    dest[0] = C_DATA;
    dest[1] = sequence_number_N;

    uint8_t l1 = data_size % 256;
    uint8_t l2 = (data_size - l1) / 256;

    dest[2] = l2;
    dest[3] = l1;
```

```
    memcpy(dest + PACKAGE_H_SIZE, data, data_size);

    sequence_number_N = (sequence_number_N + 1) % 256; //update sequence numbe
r

    return PACKAGE_H_SIZE + data_size;
}

int isEndPackage(uint8_t *package, uint8_t *start_package, unsigned int start_
package_length)
{
    for (unsigned int i = 1; i < start_package_length; i++)
    {
        // printf("pack: %x vs start pack: %x\n", package[i], start_package[i]
);
        if (package[i] != start_package[i])
            return FALSE;
    }
    return TRUE;
}


int readStartPackage(uint8_t *package, unsigned int package_size, uint8_t *fil
e_size, uint8_t *file_name)
{
    // printf("C = %x = %x?\n", package[0], C_START);
    if (package[0] != C_START)
        return -1;

    //printf("package size - %d\n", package_size);

    unsigned int curr_pack_byte = 1;
    while (curr_pack_byte < package_size)
    {
        uint8_t type = package[curr_pack_byte++];
        //printf("%x | ", package[curr_pack_byte - 1]);
        uint8_t length = package[curr_pack_byte++];
        //printf("%x | ", package[curr_pack_byte - 1]);

        for (unsigned i = 0; i < length; i++)
        {
            if (type == T_SIZE)
                file_size[i] = package[curr_pack_byte++];
            else if (type == T_NAME)
                file_name[i] = package[curr_pack_byte++];
```

```
            //printf("%x | ", package[curr_pack_byte - 1]);
        }
        //printf("\n");
    }
    //printf("\n");
    return curr_pack_byte;
}

int readDataPackage(uint8_t *package, uint8_t *dest, uint8_t expected_N){

    uint8_t read_sequence_N = package[1];
    printf("Received Sequence N: %d\n", read_sequence_N);

    if (expected_N != read_sequence_N){
        return -1;
    }

    uint8_t l2 = package[2];
    uint8_t l1 = package[3];
    int data_length = 256 * l2 + l1;
    for (int i = 0; i < data_length; i++)
    {
        dest[i] = package[i + 4];
        // printf("%x", dest[i]);
    }
    // printf("\n");



    return data_length;
}

int readPackage(uint8_t *package, uint8_t *dest, uint8_t expected_N, uint8_t *
start_package, unsigned int start_package_length)
{

    if (package[0] == C_DATA)
    {
        printf("Received Data Package\n");
        return readDataPackage(package, dest, expected_N);
    }
    else if (package[0] == C_END)
    {
        if (isEndPackage(package, start_package, start_package_length))
        {
            printf("Received Valid End Package\n");
            return VALID_END_PACKAGE;
        }
```

```
        else
        {
            printf("Received Invalid End Package\n");
            return INVALID_END_PACKAGE;
        }
    }
    printf("Received Invalid Package\n");
    return INVALID_PACKAGE;
}

void display_completion(unsigned int file_size, unsigned int curr_size){

    float file_in_packages = (file_size / ((unsigned ) PACKAGE_DATA_SIZE));
    int percentage = ((curr_size / file_in_packages) * 100);
    int squares = percentage / 5;
    printf("%d%% completed => [ ", percentage);
    for (int i = 0; i < 20; i++){
        if (i < squares)
           printf("+ ");
        else printf("- ");
    }
    printf("]\n");
}
```

noncanonical.c

```
/*Non-Canonical Input Processing*/

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include "llfunctions.h"
#include "app_package_handling.h"

#include <sys/time.h>
#include <time.h>



#define _POSIX_SOURCE 1 /* POSIX compliant source */

volatile int STOP = FALSE;
```

```c
int isFileSizeExpected(int fd, unsigned int file_size){

  struct stat st;
  fstat(fd, &st); //retrieves size of file
  return (st.st_size == file_size);
}

int getLastCharPos(char *str, char character){

  unsigned int str_size = strlen(str);
  for (int i = str_size - 1; i >= 0; i--){
    if (str[i] == character)
      return i;
  }
  return -1;
}

//creates a file with the requested name, if it doesn't already exist. Creates
 a copy otherwise. Returns file descriptor
int createReceivedFile(char *file_name){

  int file;
  int i = 1;
  char extension[EXTENSION_MAX_SIZE];
  char file_name_no_extension[FILE_NAME_MAX_SIZE];

  int last_dot_pos = getLastCharPos(file_name, '.');//position of last '.'
  if (last_dot_pos > 0)//if it has an extension
  {
    strcpy(extension, file_name + last_dot_pos); //get extension
    memcpy(file_name_no_extension, file_name, last_dot_pos); //get file name w
ithout extension
    file_name_no_extension[last_dot_pos] = '\0';
  }
  else strcpy(file_name_no_extension, file_name);

  while ((file = open(file_name, O_WRONLY | O_CREAT | O_EXCL | O_APPEND, 0777)
) < 0)
  {
    if (errno == EEXIST)
    {
      if (last_dot_pos > 0) //if it has extension
        sprintf(file_name, "%s(%d)%s", file_name_no_extension, i, extension);
      else
        sprintf(file_name, "%s(%d)", file_name_no_extension, i);
      if (strlen(file_name) < FILE_NAME_MAX_SIZE){
        //if the size does not exceed the maximum file
```

```
        i++;
      }
      else {
        perror("file name too big");
        exit(-1);
      }
    }
    else
    {
      perror("error on creating file");
      exit(-1);
    }
  }
  return file;
}

int main(int argc, char **argv)
{
  int fd; //, c, res = 0;
  unsigned char buf[D_MAX_SIZE];

  /*unsigned char bufo[14] = {FLAG_STUFF, 0x02, 0x20, 0x40, ESC_OCT, F_FLAG, F
_FLAG, ESC_OCT, 0x20, 0x30, 0x10, ESC_OCT, 0x20, 0x30};
  unsigned char bufon[3] = {0x0, 0x0E, 0x70};

  sendICommand(1, bufon, 3);*/

  if ((argc < 2) ||
      ((strcmp("/dev/ttyS0", argv[1]) != 0) &&
       (strcmp("/dev/ttyS1", argv[1]) != 0) &&
       (strcmp("/dev/ttyS2", argv[1]) != 0) &&
       (strcmp("/dev/ttyS4", argv[1]) != 0)))
  {
    printf("Usage:\tnserial SerialPort\n\tex: nserial /dev/ttyS1\n");
    exit(1);
  }

  /*
    Open serial port device for reading and writing and not as controlling tty
    because we don't want to get killed if linenoise sends CTRL-C.
  */

  fd = llopen(argv[1], RECEIVER);
  if (fd == 0)
  {
    perror(argv[1]);
    exit(-1);
  }
```

```
  //serialRead(fd, buf);
  // while (1)
  // {
  //   llread(fd, buf);
  // }

  //clock test

  struct timeval start, stop;
  gettimeofday(&start, NULL); // get initial time-stamp
  //random erros generation
  srand(time(NULL));

  /*
    O ciclo WHILE deve ser alterado de modo a respeitar o indicado no gui□o
  */

  int bytes_read;
  enum appState state = START;
  uint8_t data_package[D_MAX_SIZE];
  uint8_t start_package[D_MAX_SIZE];
  int s_pack_size;
  char file_name[FILE_NAME_MAX_SIZE];
  unsigned int file_size;
  int file = -1;

  uint8_t expected_N = 0;
  int number_of_N_groups = 0; //incremented each time N reaches 255 (N is mod
256) (each group corresponds to 256 packets)

  while (STOP == FALSE)
  {
    bytes_read = llread(fd,data_package); //get message from serial port
    if (bytes_read != DISCARD) //if the data is new and has no errors
    {
      //read the serial port message and store it in data_package
      if (state == START)
      {
        s_pack_size = readStartPackage(data_package, bytes_read, (uint8_t *) &
file_size, (uint8_t *) file_name);
        if (s_pack_size > 0)
        {
          file = createReceivedFile(file_name); //create the file to be receiv
ed
          memcpy(start_package, data_package, s_pack_size); //store the start
package for future validation
          stateMachineApp(&state); //start reading data
```

```
        }
      }
      else if (state == DATA)
      {
        int read_result = readPackage(data_package, buf, expected_N, start_pac
kage, s_pack_size);
        if (read_result > 0){
          //is valid data package
          if (write(file, buf, read_result) < 0)
          {
            perror("writing to file");
            exit(2);
          }
          //
          display_completion(file_size, expected_N + 256 * number_of_N_groups)
;
          if (expected_N == 255)
            number_of_N_groups++;
          expected_N = (expected_N + 1) % 256;
          //
        }
        else if (read_result == VALID_END_PACKAGE)
          stateMachineApp(&state); //move on to END phase
        else {
          printf("UNEXPECTED/INVALID PACKAGE\n");
        }
      }

      if (state == END)
      {
        stateMachineApp(&state);
      }
    }

    if (state == STOP_)
      STOP = TRUE;
  }


  gettimeofday(&stop, NULL); // get final time-stamp


  if (llclose(fd, RECEIVER) < 0)
  {
    printf("Timed out while closing port\n");
  }

  if (!isFileSizeExpected(file, file_size))
```

```
  {
    printf("Received file size is different than expected!!!\n");
  }

  close(file);




  double seconds = (double)(stop.tv_sec - start.tv_sec);

  double useconds = (seconds * 1000000) + stop.tv_usec - (start.tv_usec);

  double t_s = useconds / 1000000;
  // subtract time-stamps and
  // multiply to get elapsed
  // time in ns
  printf("%f, time elapsed\n", t_s);


  return 0;
}
```

writenoncanonical.c

```
/*Non-Canonical Input Processing*/

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include "llfunctions.h"
#include "app_package_handling.h"

#define MODEMDEVICE "/dev/ttyS1"
#define _POSIX_SOURCE 1 /* POSIX compliant source */

volatile int STOP = FALSE;

int sendMessage(int bytes_written, int fd, uint8_t *data_package)
{

  int resend = FALSE;
```

```
  if (bytes_written > 0)
  {
    int write_result = llwrite(fd, data_package, bytes_written);
    if (write_result == RESEND_RET)
      resend = TRUE;
    else if (write_result == TIMEOUT_RET)
    {
      printf("TIMED OUT\n");
      exit(2);
    }
    // printf("message sent - %d bytes\n", write_result);
  }
  return resend;
}

int fileDataReading(int fd, uint8_t *data_package, enum appState *state)
{

  uint8_t buf[PACKAGE_DATA_SIZE];
  int bytes_read = read(fd, buf, PACKAGE_DATA_SIZE);

  // for (unsigned int i = 0; i < bytes_read; i++)
  // {
  //   printf("%x", buf[i]);
  // }
  // printf("\n");

  if (bytes_read < 0)
  {
    perror("file reading");
    exit(1);
  }
  else if (bytes_read == 0)
  {
    //if it reaches the end of the file, go to the END state
    stateMachineApp(state);
  }
  else
  {
    bytes_read = buildDataPackage(bytes_read, buf, data_package);
  }
  return bytes_read;
}

int setupControlPackage(int fd, uint8_t *data_package, uint8_t c, enum appStat
e *state, char *file_name, unsigned int name_size)
{
```

```
  struct stat st;
  fstat(fd, &st); //retrieves size of file
  int bytes_written = buildControlPackage(c, st.st_size, (uint8_t *) file_name
, name_size, data_package);
  stateMachineApp(state); //if Start, goes to DATA state; if END, goes to stop
 state
  return bytes_written;
}

int main(int argc, char **argv)
{
  int fd; //, c, res;
  unsigned char buf[D_MAX_SIZE];
  //int i, sum = 0, speed = 0;

  if ((argc < 3) ||
      ((strcmp("/dev/ttyS0", argv[1]) != 0) &&
       (strcmp("/dev/ttyS1", argv[1]) != 0) &&
       (strcmp("/dev/ttyS2", argv[1]) != 0)))
  {
    printf("Usage:\tnserial SerialPort FileName\n\tex: nserial /dev/ttyS1 ping
uim.gif\n");
    exit(1);
  }

  //file name and respective size
  unsigned file_name_length = strlen(argv[2]);
  if (file_name_length > ORIGINAL_FILE_MAX_SIZE ){
    printf("File name is too big\n");
    exit(4);
  }

  //open file to be read
  int file = open(argv[2], O_RDONLY, 0777);
  if (file < 0)
  {
    perror("error on opening requested file");
    exit(-1);
  }
  //

  /*
    Open serial port device for reading and writing and not as controlling tty
    because we don't want to get killed if linenoise sends CTRL-C.
  */

  fd = llopen(argv[1], TRANSMITTER);
  if (fd == 0)
```

```
  {
    perror(argv[1]);
    exit(-1);
  }
  else if (fd == TIMEOUT_RET)
  {
    printf("Timed out while opening port\n");
    exit(2);
  }

  /*
    O ciclo FOR e as instru??es seguintes devem ser alterados de modo a resp
eitar
    o indicado no gui?o
  */

  int resend = FALSE;
  int bytes_written;
  enum appState state = START;
  uint8_t data_package[D_MAX_SIZE];

  while (STOP == FALSE)
  {
    if (!resend)
    {
      //if the previous message was received sucessfully, build the next one
      if (state == START)
      {
        bytes_written = setupControlPackage(file, data_package, C_START, &stat
e, argv[2], file_name_length);
      }
      else if (state == DATA)
      {
        bytes_written = fileDataReading(file, data_package, &state);
        //printf("%d - BYTES WRITTEN\n", bytes_written);
      }
      else if (state == END)
      {
        bytes_written = setupControlPackage(file, data_package, C_END, &state,
 argv[2], file_name_length);
      }
    }
    resend = sendMessage(bytes_written, fd, data_package);
    if (!resend && state == STOP_) //if it reached the end and the END Package
 has been received sucessfully
      STOP = TRUE;
  }
```

```
  //close
  if (llclose(fd, TRANSMITTER) < 0)
  {
    printf("Timed out while closing port\n");
  }

  close(file);

  return 0;
}
```

12. Anexo II – Tabelas

Variação da Capacidade de Ligação

Total bytes	10968
Total bits	87744
Package Size	255

C		T(s)	R(bit/s)	S(R/C)	S MEDIO
1200		97,705888	898,042091	0,748368	0,74837
1200		97,706033	898,040759	0,748367	
1200		97,705034	898,049941	0,748375	
1200		97,705932	898,041687	0,748368	
1800		65,139124	1347,024562	0,748347	0,748345
1800		65,139052	1347,026051	0,748348	
1800		65,139755	1347,011514	0,74834	
1800		65,139332	1347,020261	0,748345	
2400		50,222229	1747,114808	0,727965	0,727965
2400		50,222272	1747,113313	0,727964	
2400		50,222326	1747,111434	0,727963	
2400		50,222001	1747,12274	0,727968	
4800		25,115359	3493,639092	0,727841	0,727814
4800		25,115475	3493,622956	0,727838	
4800		25,115449	3493,626572	0,727839	
4800		25,11893	3493,142423	0,727738	
9600		12,561549	6985,125799	0,727617	0,727614
9600		12,56166	6985,064076	0,727611	
9600		12,561392	6985,213104	0,727626	
9600		12,561814	6984,978443	0,727602	
19200		6,284633	13961,674453	0,727171	0,727149
19200		6,284739	13961,438971	0,727158	
19200		6,284883	13961,119085	0,727142	
19200		6,285007	13960,843639	0,727127	
38400		3,146729	27884,193396	0,726151	0,726528
38400		3,219702	27252,21154	0,709693	
38400		3,108139	28230,397675	0,735167	
38400		3,108417	28227,872901	0,735101	

Variação do Tempo de Propagação (T_prop)

Total bytes	10968
Total bits	87744
Package Size	255
Baudrate (C)	38400

T_prop(s)	T(s)	R(bit/s)	S(R/C)	S MEDIO
0	3,146729	27884,193396	0,726151	0,699662
0	3,219702	27252,21154	0,709693	
0	3,146633	27885,044109	0,726173	
0	3,589213	24446,584808	0,63663	
1	45,226076	1940,119678	0,050524	0,050523
1	45,22889	1939,99897	0,050521	
1	45,229982	1939,952132	0,05052	
1	45,221825	1940,302055	0,050529	
2	90,238982	972,351395	0,025322	0,025321
2	90,235119	972,393021	0,025323	
2	90,234355	972,401254	0,025323	
2	90,262077	972,102603	0,025315	

Variação do tamanho dos pacotes de dados

Total bytes	10968
Total bits	87744
Baudrate (C)	38400

Package S	T(s)	R(bit/s)	S(R/C)	S MEDIO
40	4,434645	19786,025713	0,515261	0,514873
40	4,439087	19766,226704	0,514745	
40	4,439213	19765,665671	0,514731	
40	4,439003	19766,600743	0,514755	
80	3,622802	24219,927007	0,630727	0,630751
80	3,622316	24223,176553	0,630812	
80	3,622537	24221,698771	0,630773	
80	3,623013	24218,516467	0,630691	
120	3,367995	26052,295208	0,678445	0,678514
120	3,367383	26057,030044	0,678568	
120	3,367639	26055,04925	0,678517	
120	3,367598	26055,366466	0,678525	
160	3,245112	27038,820232	0,704136	0,704012
160	3,245848	27032,689146	0,703976	
160	3,245888	27032,356015	0,703968	
160	3,245885	27032,380999	0,703968	
200	3,169515	27683,730792	0,72093	0,720952
200	3,169441	27684,37715	0,720947	
200	3,169723	27681,914161	0,720883	
200	3,169009	27688,151091	0,721046	
255	3,146729	27884,193396	0,726151	0,726528
255	3,219702	27252,21154	0,709693	
255	3,108139	28230,397675	0,735167	
255	3,108417	28227,872901	0,735101	

Variação de FER

Total bytes	10968
Total bits	87744
Package Size	255
Baudrate (C)	38400

BCC1

BCC1	T(s)	R(bit/s)	S(R/C)	S MEDIO
0	3,146729	27884,193396	0,726151	0,699662
0	3,219702	27252,21154	0,709693	
0	3,146633	27885,044109	0,726173	
0	3,589213	24446,584808	0,63663	
2	3,809237	23034,534212	0,599858	0,322536
2	7,49916	11700,510457	0,304701	
2	10,543155	8322,366502	0,216728	
2	13,532164	6484,10705	0,168857	
4	10,609489	8270,332341	0,215373	0,219478
4	16,698142	5254,716363	0,136842	
4	7,595569	11551,998277	0,300833	
4	10,161645	8634,822413	0,224865	
6	10,854174	8083,89473	0,210518	0,178769
6	20,007201	4385,620957	0,114209	
6	23,148625	3790,462716	0,09871	
6	7,835044	11198,916049	0,291638	
8	13,931043	6298,451595	0,164022	0,139019
8	14,028543	6254,676626	0,162882	
8	19,992404	4388,866892	0,114293	
8	19,890456	4411,361911	0,114879	
10	19,313945	4543,038722	0,118308	0,158478
10	16,947312	5177,458231	0,13483	
10	7,802289	11245,930521	0,292863	
10	25,99233	3375,765081	0,087911	

BCC2

BCC2	T(s)	R(bit/s)	S(R/C)	S MEDIO
0	3,146729	27884,193396	0,726151	0,699662
0	3,219702	27252,21154	0,709693	
0	3,146633	27885,044109	0,726173	
0	3,589213	24446,584808	0,63663	
2	4,370563	20076,132068	0,522816	0,535086
2	4,074575	21534,51587	0,560795	
2	4,574175	19182,475528	0,499544	
2	4,100937	21396,085821	0,55719	
4	4,586252	19131,962221	0,498228	0,506427
4	4,824915	18185,605342	0,473583	
4	4,2982	20414,126844	0,531618	
4	4,375056	20055,514718	0,522279	
6	4,720032	18589,704477	0,484107	0,462508
6	4,990139	17583,478136	0,457903	
6	4,830078	18166,166261	0,473077	
6	5,253535	16701,896913	0,434945	
8	4,418142	19859,932071	0,517186	0,46872
8	5,32522	16477,065736	0,42909	
8	4,303248	20390,179697	0,530994	
8	5,746857	15268,171802	0,397609	
10	4,85291	18080,697973	0,470852	0,463948
10	4,537821	19336,152748	0,503546	
10	5,016693	17490,406529	0,455479	
10	5,364897	16355,206819	0,425917	

Total bytes	10968
Total bits	87744
Package Size	255
Baudrate (C)	38400