FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Ligação de Dados

Final Report
RC 2022/23

T02

Diogo Babo (up202004950@fe.up.pt)

João Oliveira (up202004407@fe.up.pt)

Índice

Sumário:

Este relatório incide sobre o primeiro trabalho prático da cadeira de Redes de Computadores, sendo o objetivo do mesmo o desenvolvimento de uma aplicação que permitisse transferir ficheiros entre dois computadores distintos através de uma porta de série.

A aplicação desenvolvida satisfaz o objetivo proposto e também lida com os eventuais erros que possam acontecer na transferência dos ficheiros entre os computadores.

Introdução:

O objetivo do trabalho é implementar um protocolo de ligação de dados, sendo que este relatório tem a finalidade de explicar a nossa abordagem no desenvolvimento do mesmo. O relatório está dividido em 9 partes:

- **Arquitetura** - blocos funcionais e interfaces
- **Estrutura do Código** - APIs, principais estruturas de dados, principais funções e a sua relação com a arquitetura.
- **Casos de uso principais** - identificação; sequências de chamada de funções.
- **Protocolo de ligação lógica** - identificação dos principais aspetos funcionais e descrição da estratégia de implementação
- **Protocolo de aplicação** - identificação dos principais aspetos funcionais e descrição da estratégia de implementação
- **Validação** - descrição dos testes efetuados com apresentação quantificada dos resultados.
- **Eficiência do protocolo de ligação de dados** - caracterização estatística da eficiência do protocolo
- **Conclusões** - síntese da informação apresentada nas seções anteriores; reflexão sobre os objetivos de aprendizagem alcançados.
- **Anexos** - código fonte, testes e resultados

Arquitetura:

O nosso projeto foi desenvolvido em duas componentes distintas, a Application Layer e Link Layer**.**

A **Application Layer** é responsável por receber os diferentes argumentos do utilizador, como o nome do ficheiro e a porta de série, e é nesta mesma camada onde se dá a leitura do ficheiro original e a escrita para o novo ficheiro.

Por outro lado, é na **Link Layer** que foram desenvolvidas as funções para abrir e fechar a ligação da porta de série assim como escrever e ler da mesma.

Estrutura do Código:

A aplicação está dividida em 4 ficheiros, sendo que cada um destes tem associado um header file próprio, e ainda os ficheiros **main.c** e **macros.h**.

Application Layer - application_layer.c

Funções Principais:

- **applicationLayer -** Função responsável por abrir e fechar a ligação com a porta de série. Tem funcionalidades diferentes tendo em conta o papel (receptor/transmissor). Sendo um **transmissor**, dá-se o ciclo em que se lê o ficheiro e se montam os pacotes de controlo e dados, usando as funções **mountCtrlPacket** & **mountDataPacket.** Em cada iteração do ciclo, monta-se a trama de informação e envia-se para a porta de série através da função **llwrite**. No caso de ser um **receptor**, há um ciclo para ler as tramas de informação, dá-se o parse do pacote lido através das funções **readControlPacket** & **readDataPacket**. A cada iteração do ciclo os dados são escritos no novo ficheiro.

Link Layer - link_layer.c

Funções Principais:

- **llopen -** abre a porta de série recorrendo à função **openSP,** havendo a troca de tramas SET e UA
- **llwrite -** monta as tramas de informação e envia-as pela porta de série
- **llread -** lê as tramas de informação e envia a respectiva resposta (RR ou REJ)
- **llclose -** fecha a porta de série e repõe as configurações iniciais, havendo a troca de tramas DISC e UA

Funções Secundárias:

- **openSP -** função responsável por abrir a porta de série
- **alarmHandler -** Signal handler para SIGALARM
- **resetAlarm -** função responsável por resetar o counter do alarm
- **sendSet -** função responsável por enviar o SET
- **readUA -** função responsável por ler o UA
- **readSetSendUA -** função responsável por ler o SET e enviar o UA

- **sendUATransmisser -** função responsável por enviar o UA
- **sendDiscTransmisser -** função responsável por enviar o DISC
- **readAndSendDisc -** função responsável por ler o DISC e enviá-lo de novo
- **writeBadPacket -** função responsável por montar e enviar o REJ

State Machine - state_machine.c

Ficheiro que contém as funções responsáveis por dar handle à máquina de estados. Tendo em conta que as leituras da porta de série são feitas byte a byte, consoante o byte lido é chamada a função **changeState** para alterar o estado atual. Isto permite saber quando acabamos de ler a trama totalmente, por exemplo se estiver no estado BCC_OK e receber a FLAG então sabemos que a trama foi totalmente processada e o estado vai ser alterado para STOP.

Funções Principais:

- **changeState -** função responsável por alterar o estado da máquina de estados, tendo em conta o byte que recebe como argumento e o seu estado atual.

Estruturas de Dados:

enum stateMachine {START, FLAG_RCV, A_RCV, C_RCV, BCC_OK, C_INF, STOP, REJ};

Utils - utils.c

Ficheiro que contém as funções responsáveis por abrir e obter as propriedades de ficheiros (tamanho por ex), mas também por montar e dar parse dos pacotes de controlo e de dados.

Funções Principais:

- **get_file_size -** Retorna o tamanho de um ficheiro em bytes.
- **numOfBytes -** Retorna o número de bytes necessários para escrever o tamanho do ficheiro
- **mountCtrlPacket -** Monta o pacote de controlo
- **mountDataPacket -** Monta o pacote de dados
- **readControlPacket -** Processa (dá parse) aos pacotes de controlo
- **readDataPacket -** Processa (dá parse) aos pacotes de dados

Macros - macros.h

Ficheiro que contém macros para alguns valores, de modo a facilitar a leitura do código.

Casos de Uso:

Para poder utilizar a nossa aplicação, esta deve ser compilada a partir do ficheiro **MakeFile** fornecido da seguinte forma:

- **Transmissor:** make run_tx
- **Receptor:** make run_rx

Caso o utilizador queira alterar tanto a porta de série como o ficheiro a ser enviado, estes parâmetros devem ser alterados no próprio **MakeFile**, e depois apenas é necessário compilar a aplicação de novo para esta funcionar com os novos parâmetros.

O receptor deve ser inicializado primeiro, estando então à espera que o transmissor comece a transmissão.Se o transmissor for inicializado primeiro, este irá tentar enviar as mensagens para o receptor e ao final do número de tentativas estipuladas o programa irá terminar caso não receba a resposta esperada.

Assim que ambos estejam conectados, o transmissor vai começar a enviar a informação do ficheiro para o receptor, e o programa irá mostrar na consola mensagens sobre o envio da informação como possíveis mensagens de erros que poderão eventualmente acontecer.

Protocolo de ligação lógica:

O protocolo de ligação lógica é responsável por iniciar e terminar a ligação com a porta de série, assim como pela escrita e a leitura de informação através da mesma.

llopen():

Esta função é responsável por abrir a ligação entres os computadores com a porta de série. Recebe como argumento a struct do tipo **LinkLayer**, esta que guarda informação sobre o número de retransmissões, timeout, o baudrate, a role, o nome do ficheiro e por fim a porta de série a ser usada. Ela começa por chamar a função **openSP**, que vai abrir a porta de série para leitura e escrita tendo em conta o path da mesma recebida na struct. Além disso, vai também configurar a porta de série com o **VTIME** a 0.1 e o **VMIN** a 0 para esta não bloquear quando não lê nenhum caractere.

Dependendo do role (transmissor/receptor), que é tratado através de um switch case, são chamadas as funções **sendSet** no caso do transmissor e **readSetSendUA** no caso do receptor.

A função **sendSet** é responsável por montar a trama de supervisão SET, mas também por receber o **UA** vindo do receptor. Caso não receba o UA, ele vai reenviar o SET o número de vezes estipulado (nTries) e termina o programa ao fim desse número de tentativas caso acabe por não receber o UA.

A função **readSetSendUA** é responsável por receber o SET vindo do transmissor, e após a leitura do mesmo envia o UA.

llwrite():

Esta função é responsável por enviar para a porta de série os pacotes de dados,sob forma de uma trama de informação. Recebe como argumentos um array de caracteres e o comprimento do mesmo.

Inicialmente a função começa por montar o header das tramas I (4 bytes), começando pela FLAG, o campo de endereço, campo de controlo (que depende do número de sequência) e o BCC1 que é um XOR entre o campo de endereço e o de controlo. De seguida, calcula-se o BCC2 através de um XOR com os dados e faz-se o **Byte Stuffing** quer do BBC2 quer dos dados. Monta-se a trama com os dados e com o bcc2 stuffed, e por fim concatena-se a FLAG montando então a trama I.

A função envia esta trama e fica à espera de uma resposta (RR ou REJ) por parte do receptor. Na eventualidade de esta ser um REJ, o número de sequência não é atualizado, e a função tenta enviar de novo até o número de retransmissões ser atingido ou receber um RR.

llread():

Esta função é responsável por ler da porta de série os pacotes de dados,sob forma de uma trama de informação. Recebe como argumentos um array de caracteres, onde se vão armazenar os dados recebidos.

A função começa por ler a trama I da porta de série, byte a byte, sabendo que chegou ao fim da trama quando alcança o estado STOP. Após ler tudo, dá-se o **Byte Destuffing** tanto nos dados como no BCC2 e calcula-se um novo BCC2 com os dados recebidos. Verifica-se se o BCC2 calculado bate de acordo com o BCC2 recebido, caso isso aconteça então o array recebido com argumento é preenchido e a função envia um RR, caso contrário envia um REJ e os dados são descartados.

llclose():

Esta função é responsável por terminar a ligação entres os computadores com a porta de série.

Com uma estrutura semelhante à do **llopen**, esta função possui um switch case que executa diferentes funções dependendo da role (transmissor/receptor).

Na perspetiva do transmissor são executadas as funções **sendDiscTransmisser** e **sendUATransmisser**. A primeira trata de enviar a trama de supervisão DISC e também ler a mesma trama vinda do receptor. Caso isto não aconteça, reenvia o DISC o número de vezes estipulado, dando erro caso acabe por não a receber. A segunda função é apenas responsável por montar a trama de supervisão UA e enviá-la para a porta de série.

Na perspetiva do receptor são executadas as funções **readAndSendDisc** e **readUA**. A primeira função trata de ,como o próprio nome indica, de receber uma trama de supervisão DISC e enviar a mesma trama para o transmissor. A segunda função, por outro lado, apenas é responsável por receber uma trama UA.

Além disso, a função repõe as configurações da porta de série que estavam inicialmente e é terminada a ligação com a mesma.

Protocolo de Aplicação:

O protocolo de ligação lógica é responsável pela leitura do ficheiro original e pela escrita no ficheiro final, assim como por montar e dar parse tanto aos pacotes de controlo como aos de dados. (Funções estas que estão implementadas no utils.c)

mountCtrlPacket():

Esta função é responsável por montar um pacote de controlo. Recebe como argumentos um array de caracteres, o tipo de pacote como um inteiro, 0 se for de início e 1 se for de fim.

Recebe também o nome do ficheiro e o seu tamanho em bytes. No final, vai armazenar no array recebido nos argumentos o pacote de controlo, retornando o tamanho do mesmo.

mountDataPacket():

Esta função é responsável por montar um pacote de dados. Recebe como argumentos um array de caracteres, o número de sequência, o tamanho dos dados e um array com os dados . No final, vai armazenar no array recebido nos argumentos o pacote de dados, retornando o tamanho do mesmo.

readControlPacket():

Esta função é responsável por dar parse de um pacote de controlo, retornando o tamanho do ficheiro e o seu nome nos argumentos (que são passados como referência).

readDataPacket():

Esta função é responsável por dar parse de um pacote de dados, retornando os dados pacote (descartando o header) num array de caracteres que é passado nos argumentos por referência.

applicationLayer():

A função começa por colocar na struct do tipo LinkLayer os valores recebidos como argumentos vindos da função **main**.

A função abre então a ligação à porta série tanto para o transmissor como para o receptor, através da função **llopen**. Existe um switch case, que vai diferentes procedimentos dependendo da role.

No caso de ser um transmissor, o programa vai começar por abrir o ficheiro original e obter o seu tamanho. Vai montar um pacote de controlo do tipo START chamando a função **mountCtrlPacket** e vai escrevê-lo na porta de série através da função **llwrite**. Após o envio deste pacote, vai começar o ciclo de leitura do ficheiro em que a cada iteração se vai ler um tamanho predefinido (**MAX_PAYLOAD_SIZE**) enviando o respectivo pacote de dados através do **llwrite**. Quando a leitura do ficheiro se der por concluída, vai se montar um pacote de controlo do tipo END, chamando a função **mountCtrlPacket** escrevendo-o na porta de série através da função **llwrite**.

No caso do receptor, abre-se o ficheiro para o qual vamos escrever e dá-se um ciclo de leitura em que a cada iteração se lê da porta de série chamando a função **llread**, e verifica-se o tipo de pacote. Caso seja um pacote de dados, então a aplicação dá parse do mesmo chamando a função **readDataPacket()** e escreve no ficheiro de destino os dados. A condição de paragem do ciclo, é quando se lê um pacote de controlo cujo tipo é END.

Finalmente, a aplicação chama a função **llclose** e termina a ligação com a porta de série.

Validação:

Foram realizados alguns testes para garantir o bom funcionamento da aplicação:

- Interrupção da porta de série a meio da execução do programa;
- Fazer ruído na porta de série para verificar que o programa atuava como pretendido;
- Enviar ficheiros diferentes;

- Enviar o ficheiro variando o tamanho do pacote - 125,250,500,1000,2000 e 4000 bytes
- Enviar o ficheiro variando o Baudrate - 9600,19200,38400,57600 e 115200
- Enviar o ficheiro gerando erros no BCC2 - 0%, 2%, 5%, 10% e 20%

Eficiência do protocolo de ligação de dados:

Todos os testes e os seus respectivos resultados vêm como anexo no final deste relatório.

Variação do Tamanho do Pacote:

Através da variação do tamanho dos pacotes é possível verificar que para pacotes de tamanho superior a eficiência é superior, uma vez que vão ser precisas menos iterações para enviar o ficheiro.

Variação do Baudrate:

Através da variação da Baudrate é possível verificar que há uma variação muito mínima na eficiência, não sendo então significativa. No entanto, para baudrates menores a eficiência é ligeiramente superior.

Variação do Erro no BCC2:

Através da variação dos erros no BCC2 é possível verificar que a eficiência diminui com o aumento da percentagem de erros, uma vez que vão haver retransmissões e o número de iterações para enviar o ficheiro vai ser superior

Conclusão:

Neste relatório descrevemos a nossa abordagem face ao desafio proposto, a forma de como organizamos o código, as funções utilizadas, os resultados e os testes realizados.

Um dos objetivos do trabalho era perceber a independência entre as duas camadas, a da ligação e a de aplicação. Sendo que a camada de ligação trata da escrita e leitura de informação através da porta de série, enquanto que a da aplicação usa as funções implementadas na outra camada, contudo sem ter conhecimento sobre o seu funcionamento.

Foi possível cumprir o objetivo proposto, tendo implementado ambas as camadas com sucesso, obtendo uma eficiência de cerca de 70%. Eficiência esta que poderia ser melhorada, através de uma melhor abordagem e implementação de algumas funções.

Anexo I - Código Fonte

main.c

```
#include <stdio.h>
#include <stdlib.h>

#include "application_layer.h"

#define BAUDRATE 9600
#define N_TRIES 3
#define TIMEOUT 4

// Arguments:
//   $1: /dev/ttySxx
//   $2: tx | rx
//   $3: filename
int main(int argc, char *argv[])
{
    if (argc < 4)
    {
        printf("Usage: %s /dev/ttySxx tx|rx filename\n", argv[0]);
        exit(1);
    }
    const char *serialPort = argv[1];
    const char *role = argv[2];
    const char *filename = argv[3];
    printf("Starting link-layer protocol application\n"
           "  - Serial port: %s\n"
           "  - Role: %s\n"
           "  - Baudrate: %d\n"
           "  - Number of tries: %d\n"
           "  - Timeout: %d\n"
           "  - Filename: %s\n",
           serialPort,
           role,
           BAUDRATE,
           N_TRIES,
           TIMEOUT,
           filename);

    applicationLayer(serialPort, role, BAUDRATE, N_TRIES, TIMEOUT,
filename);

    return 0;
}
}
```

application_layer.h

```
// Application layer protocol header.
// NOTE: This file must not be changed.

#ifndef _APPLICATION_LAYER_H_
#define _APPLICATION_LAYER_H_

// Application layer main function.
// Arguments:
//   serialPort: Serial port name (e.g., /dev/ttyS0).
//   role: Application role {"tx", "rx"}.
//   baudrate: Baudrate of the serial port.
//   nTries: Maximum number of frame retries.
//   timeout: Frame timeout.
//   filename: Name of the file to send / receive.
void applicationLayer(const char *serialPort, const char *role, int
baudRate,
                      int nTries, int timeout, const char *filename);

#endif // _APPLICATION_LAYER_H_
```

application_layer.c

```
#include "application_layer.h"
#include "link_layer.h"
#include "utils.h"

extern int valid;
extern int nr;

void applicationLayer(const char *serialPort, const char *role, int
baudRate,
                      int nTries, int timeout, const char *filename)
{
    LinkLayer str;
    strcpy(str.serialPort,serialPort);
    str.nRetransmissions = nTries;
    str.timeout = timeout;
    str.baudRate = baudRate;

    if(strcmp(role,"tx") == 0) {
        str.role = LlTx;
    }
    else if(strcmp(role,"rx") == 0) {
        str.role = LlRx;
    }
```

```
llopen(str);
resetAlarm();
if(str.role == LlTx) {
    unsigned char pack[MAX_PAYLOAD_SIZE];
    unsigned char buf[MAX_PAYLOAD_SIZE];

    FILE* src = fopen(filename,"r");
    int filesz = get_file_size(src);

    int ctrlsize = mountCtrlPacket(&pack,0,filename,filesz);
    llwrite(&pack,ctrlsize);
    int bytes_r;
    int sq = 0;
    resetAlarm();
    while(bytes_r = fread(buf,1,MAX_PAYLOAD_SIZE - 4,src)) {
        unsigned char data[MAX_PAYLOAD_SIZE];
        int alo = mountDataPacket(&data,sq,bytes_r,buf);
        llwrite(&data,alo);
        resetAlarm();
        fprintf(stderr, "Data Packet Sent: Size: %d\n", alo);
        sq++;
    }

    fprintf(stderr, "Control Packet End Sent:\n");
    ctrlsize = mountCtrlPacket(&pack,1,filename,filesz);
    llwrite(&pack,ctrlsize);
    resetAlarm();
}
else if(str.role == LlRx) {
    int* filesz;
    char* filen;
    unsigned char pack[MAX_PAYLOAD_SIZE + 7];
    unsigned char data[MAX_PAYLOAD_SIZE + 7];

    int sz = llread(&pack);

    FILE* dest = fopen(filename,"w");
    int ns = 0;

    while(1) {
        int sq;
        int lastnr = nr;
        do{
            sz = llread(&pack);
            printf("reading package\n");
```

```
            }while (sz == -1);
            fprintf(stderr,"New Packet:\n");
            if(pack[0] == 0x03) {
                fprintf(stderr,"Read End\n");
                break;
            }
            else if(pack[0] == 0x01){
                sz = readDataPacket(&data,&pack,&sq);
                printf("\n\n");
                /*for(int i = 0; i < sz; i++) {
                    fprintf(stderr,"\\%02x", data[i]);
                }*/
                fwrite(data,1,sz,dest);
            }
        }
    }
    llclose(0);
}
```

link_layer.h

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


#ifndef _LINK_LAYER_H_
#define _LINK_LAYER_H_

typedef enum
{
    LlTx,
    LlRx,
} LinkLayerRole;

typedef struct
{
    char serialPort[50];
```

```
    LinkLayerRole role;
    int baudRate;
    int nRetransmissions;
    int timeout;
} LinkLayer;

// SIZE of maximum acceptable payload.
// Maximum number of bytes that application layer should send to link
layer
#define MAX_PAYLOAD_SIZE 1000

// MISC
#define FALSE 0
#define TRUE 1

// Open a connection using the "port" parameters defined in struct
linkLayer.
// Return "1" on success or "-1" on error.
int llopen(LinkLayer connectionParameters);

// Send data in buf with size bufSize.
// Return number of chars written, or "-1" on error.
int llwrite(const unsigned char *buf, int bufSize);

// Receive data in packet.
// Return number of chars read, or "-1" on error.
int llread(unsigned char *packet);

// Close previously opened connection.
// if showStatistics == TRUE, link layer should print statistics in the
console on close.
// Return "1" on success or "-1" on error.
int llclose(int showStatistics);

// opens the serial port
int openSP();

// mounts and sends SET frame
int sendSet();

// reads SET and sends UA
int readSetSendUA();

// sends DISC
int sendDiscTransmisser();
```

```
// reads and sends DISC
int readAndSendDisc();

// sends UA
int sendUATransmisser();

// reads UA
int readUA();

// resets alarm counter and state
int resetAlarm();

// writes REJ to the serial port
int writeBadPacket();

#endif // _LINK_LAYER_H_
```

link_layer.c

```
#include "link_layer.h"
#include "utils.h"

// MISC
#define _POSIX_SOURCE 1 // POSIX compliant source
LinkLayer str;
enum stateMachine stOpen = START;
enum stateMachine stWrite = START;
enum stateMachine stRead = START;
enum stateMachine stClose = START;
struct termios oldtio;
struct termios newtio;
int fd;
int ns = 0;
int nr = 1;
int valid = TRUE;
int alarmEnabled = FALSE;
int alarmCount = 0;

void alarmHandler(int signal)
{
    alarmEnabled = FALSE;
    alarmCount++;
    printf("Alarm #%d\n", alarmCount);
}

////////////////////////////////////////////////
```

```
// LLOPEN
////////////////////////////////////////////////

int resetAlarm() {
    alarmCount = 0;
    alarmEnabled = FALSE;
}

int openSP() {
    fd = open(str.serialPort, O_RDWR | O_NOCTTY);

    if (fd < 0) {
        printf("error");
        return -1;
    }

    // Save current port settings
    if (tcgetattr(fd, &oldtio) == -1) {
        printf("error2");
        return -1;
    }

    memset(&newtio, 0, sizeof(newtio));
    newtio.c_cflag = str.baudRate | CS8 | CLOCAL | CREAD;
    newtio.c_iflag = IGNPAR;
    newtio.c_oflag = 0;
    newtio.c_lflag = 0;
    newtio.c_cc[VTIME] = 0.1; // Inter-character timer unused
    newtio.c_cc[VMIN] = 0;  // Read without blocking
    tcflush(fd, TCIOFLUSH);

    if (tcsetattr(fd, TCSANOW, &newtio) == -1)
        return -1;

    return fd;
}

int sendSet() {
    unsigned char buf[BUF_SIZE + 1] = {0};
    buf[0] = FLAG;
    buf[4] = FLAG;
    buf[1] = TRANS;
    buf[2] = SET;
    buf[3] = buf[1] ^ buf[2];
    (void)signal(SIGALRM, alarmHandler);
```

```
    while (stOpen != STOP && alarmCount < (str.nRetransmissions + 1))
    {
        if (!alarmEnabled)
        {
            printf("Set Written\n");
            stOpen = START;
            write(fd, buf, 5);
            alarm(str.timeout);
            alarmEnabled = TRUE;
        }

        if (read(fd, buf, 1) > 0) {
            changeState(&stOpen,buf[0],0);
            if(stOpen == STOP)
            {
                printf("Read UA\n");
                break;
            }
        }
    }
    return alarmCount;
}

int readSetSendUA() {
    unsigned char buf[BUF_SIZE + 1] = {0}; // +1: Save space for the
final '\0' char
    unsigned int n = 0;
    while (TRUE)
    {
        // Returns after 5 chars have been input
        int bytes = read(fd, buf, 1); // em bytes vai ficar o tamanho da
string
                                      // pode ser por um char
especial no fim do write, para saber quando parar de ler


        changeState(&stOpen,buf[0],0);
        if (stOpen == STOP){ // Set end of string to '\0', so we can
printf
            printf("Read SET\n");
            break;
        }
   }

    buf[0] = FLAG;
    buf[4] = FLAG;
```

```
    buf[1] = TRANS;
    buf[2] = UA;
    buf[3] = buf[1] ^ buf[2];
    int bytes = write(fd, buf, 5);
    printf("Sent UA\n");
    return 1;
}

int llopen(LinkLayer connectionParameters)
{
    str = connectionParameters;
    openSP();
    switch (str.role)
    {
    case LlTx: // write set & read UA
        sendSet();
        break;

    case LlRx: // read set & send UA
        readSetSendUA();
        break;
    default:
        break;
    }
    return 1;
}

////////////////////////////////////////////////
// LLWRITE
////////////////////////////////////////////////
int llwrite(const unsigned char *buf, int bufSize)
{
    unsigned char info[2 * MAX_PAYLOAD_SIZE];

    // montar 4 bytes iniciais
    info[0] = FLAG;
    info[1] = TRANS;
    info[2] = (ns == 0) ? 0x00 : 0x40;
    info[3] = BCC(info[1],info[2]);

    int infoSz = 0;
    int bcc_2 = 0;
    bcc_2 ^= buf[0];
    bcc_2 ^= buf[1];
    bcc_2 ^= buf[2];
    bcc_2 ^= buf[3];
```

```
for(int i = 4; i < bufSize; i++) {
    bcc_2 ^= buf[i];
}
// montar os dados
for(int i = 0; i < bufSize; i++) {
    if(buf[i] == FLAG) {
        info[4+infoSz] = ESC;
        info[5+infoSz] = ESCE;
        infoSz+= 2;
    }
    else if(buf[i] == ESC) {
        info[4+infoSz] = ESC;
        info[5+infoSz] = ESCD;
        infoSz+= 2;
    }
    else {
        info[4+infoSz] = buf[i];
        infoSz++;
    }
}
if(bcc_2 == FLAG) {
    info[4+infoSz] = ESC;
    info[5+infoSz] = ESCE;
    info[6+infoSz] = FLAG;
    infoSz+= 1;
}
else if(bcc_2 == ESC) {
    info[4+infoSz] = ESC;
    info[5+infoSz] = ESCD;
    info[6+infoSz] = FLAG;
    infoSz+= 1;
}
else {
    info[4+infoSz] = bcc_2;
    info[5+infoSz] = FLAG;
}

infoSz += 6;

unsigned char bufi[BUF_SIZE + 1] = {0};
(void)signal(SIGALRM, alarmHandler);

while (stWrite != STOP && alarmCount < (str.nRetransmissions + 1))
{
    if (!alarmEnabled || stWrite == REJ)
```

```
        {

            stWrite = START;
            fprintf(stderr, "Sent Packet Sent Type: %d\n", ns);
            int b_written = write(fd,info,infoSz);
            alarm(str.timeout);
            alarmEnabled = TRUE;
        }
        int i = 0;
        unsigned char tmp;
        if (read(fd, bufi + i, 1) > 0) {
            //fprintf(stderr,"state: %d, Buf: %02x \n",
stWrite,bufi[i]);
            //fprintf(stderr,"STate: %d", stWrite);
            changeState(&stWrite,bufi[i],0);
            if(stWrite == REJ){
                read(fd, bufi + i, 1);
                i++;
                read(fd, bufi + i, 1);
                i++;
            }
            //fprintf(stderr,"Buf: %02x \n", bufi[i]);
            i++;
            if(stWrite == STOP)
            {
                printf("Read ACK\n");
                ns = (1 + ns) % 2;
                valid = TRUE;
                break;
            }
        }
    }


    stWrite = START;
      alarmEnabled = FALSE;
    if(alarmCount >= (str.nRetransmissions + 1)){
        alarmCount = 0;
        llclose(0);
        exit(1);
    }
    return infoSz;
}


////////////////////////////////////////////////
```

```
// LLREAD
////////////////////////////////////////////////

int writeBadPacket(){
    unsigned char outbuf[6];
    outbuf[0] = FLAG;
    outbuf[4] = FLAG;
    outbuf[1] = TRANS;
    if(nr) {
        outbuf[2] = REJ_1;
    }
    else {
        outbuf[2] = REJ_0;
    }
    outbuf[3] = outbuf[1] ^ outbuf[2];
    write(fd, outbuf, 5);
    return 5;
}

int llread(unsigned char *packet)
{
    unsigned char buf[MAX_PAYLOAD_SIZE * 2];
    unsigned char initPacket[MAX_PAYLOAD_SIZE]; // com byte stuffing

    int sz = 0;
    stRead = START;

    memset(packet,0,sizeof(packet));

    while(1) {
        if(read(fd,buf + sz,1) > 0) {
            if(sz+1>MAX_PAYLOAD_SIZE * 2){
                fprintf(stderr,"END PACKET FLAG NOT FOUND: NOISE ON
SEREAL PORT\n");
                writeBadPacket();
                return -1;
            }
            changeState(&stRead,buf[sz],1);
            sz++;
            if(stRead == STOP) {
                //printf("Read All\n");
                break;
            }
        }
    }
```

```
    stRead = START;

    int bcc_2 = 0;
    if(buf[sz - 3] == ESC && buf[sz - 2] == ESCD) {
        bcc_2 = ESC;
        sz--;

    }
    else if(buf[sz - 3] == ESC && buf[sz - 2] == ESCE) {
        bcc_2 = FLAG;
        sz--;

    }else{
        bcc_2 = buf[sz-2];
    }


    for(int i = 0; i < sz - 6; i++) {
        initPacket[i] = buf[4+i];
    }



    packet[0] = initPacket[0];
    packet[1] = initPacket[1];
    packet[2] = initPacket[2];
    packet[3] = initPacket[3];

    int j = 4;
    for(int i = 4; i < sz - 6; i++) {
        if(initPacket[i] == ESC && initPacket[i+1] == ESCE) {
            packet[j] = FLAG;
            j++;
            i++;
        }
        else if(initPacket[i] == ESC && initPacket[i+1] == ESCD) {
            packet[j] = ESC;
            j++;
            i++;
        }
        else {
            packet[j] = initPacket[i];
            j++;
        }
    }
```

```
    int calBCC = 0;
    calBCC ^= packet[0];
    calBCC ^= packet[1];
    calBCC ^= packet[2];
    calBCC ^= packet[3];
    for(int i = 4; i < j; i++) {
        calBCC ^= packet[i];
    }

    //memset(buf,0,sizeof(buf));
    unsigned char outbuf[6];
    outbuf[0] = FLAG;
    outbuf[4] = FLAG;
    outbuf[1] = TRANS;


    if(calBCC == bcc_2){
        nr = (nr + 1) % 2;
        if(nr) {
            outbuf[2] = RR_1;
        }
        else {
            outbuf[2] = RR_0;
        }
    }else if(calBCC != bcc_2){
        if(nr) {
            outbuf[2] = REJ_1;
        }
        else {
            outbuf[2] = REJ_0;
        }
        printf("bad packet\n");
        outbuf[3] = outbuf[1] ^ outbuf[2];
        write(fd, outbuf, 5);
        return -1;
    }

    outbuf[3] = outbuf[1] ^ outbuf[2];

    int bytes = write(fd, outbuf, 5);

    fprintf(stderr,"send\n");

    int sz_final = j;
    return sz_final;
}
```

```
////////////////////////////////////////////////
// LLCLOSE
////////////////////////////////////////////////

int sendDiscTransmisser() {
    unsigned char buf[BUF_SIZE + 1] = {0};
    buf[0] = FLAG;
    buf[4] = FLAG;
    buf[1] = TRANS;
    buf[2] = DISC;
    buf[3] = buf[1] ^ buf[2];
    (void)signal(SIGALRM, alarmHandler);

    while (stClose != STOP && alarmCount < (str.nRetransmissions + 1))
    {
        if (!alarmEnabled)
        {
            printf("Disc Written\n");
            stClose = START;
            write(fd, buf, 5);
            alarm(str.timeout);
            alarmEnabled = TRUE;
        }

        if (read(fd, buf, 1) > 0) {
            changeState(&stClose,buf[0],0);
            if(stClose == STOP)
            {
                printf("Read Disc\n");
                break;
            }
        }
    }
    return alarmCount;
}

int readAndSendDisc() {
    unsigned char buf[BUF_SIZE + 1] = {0}; // +1: Save space for the
final '\0' char
    while (TRUE)
    {
        // Returns after 5 chars have been input
        int bytes = read(fd, buf, 1); // em bytes vai ficar o tamanho da
string
                                      // pode ser por um char
```

```
especial no fim do write, para saber quando parar de ler


        changeState(&stClose,buf[0],0);
        if (stClose == STOP){ // Set end of string to '\0', so we can
printf
            printf("Read Disc\n");
            break;
        }
    }
    buf[0] = FLAG;
    buf[4] = FLAG;
    buf[1] = RECEIVE;
    buf[2] = DISC;
    buf[3] = buf[1] ^ buf[2];
    int bytes = write(fd, buf, 5);
    printf("Send Disc\n");

    return 1;
}

int sendUATransmisser() {
    unsigned char buf[BUF_SIZE + 1] = {0};
    buf[0] = FLAG;
    buf[4] = FLAG;
    buf[1] = RECEIVE;
    buf[2] = UA;
    buf[3] = buf[1] ^ buf[2];
    write(fd, buf, 5);
    printf("Sent UA");
}

int readUA() {
    unsigned char buf[BUF_SIZE + 1] = {0}; // +1: Save space for the
final '\0' char
    unsigned int n = 0;
    while (TRUE)
    {
        // Returns after 5 chars have been input
        int bytes = read(fd, buf+n, 1); // em bytes vai ficar o tamanho
da string
                                        // pode ser por um char
especial no fim do write, para saber quando parar de ler


        changeState(&stClose,buf[n],0);
```

```
        if (stClose == STOP){ // Set end of string to '\0', so we can
printf
            printf("Read UA\n");
            break;
        }
        n++;
    }

}

int llclose(int showStatistics)
{
    alarmEnabled = 0;
    alarmCount = 0;
    switch (str.role)
    {
    case LlTx: // write DISC & read DISC
        if(sendDiscTransmisser() == 3) {return -1;}
        sendUATransmisser();
        break;

    case LlRx: // read DISC & send DISC
        readAndSendDisc();
        readUA();
        break;
    default:
        break;
    }

    sleep(1);

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


    close(fd);
    return 1;
}
```

utils.h

```c
#include "macros.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// returns file size in bytes
int get_file_size(FILE* f);

// mounts control packet
int mountCtrlPacket(unsigned char* buf, int type, char* filename, int
filesz);

// returns nr of bytes to represent a number
int numOfBytes(int filesz);

// mounts data packet
int mountDataPacket(unsigned char* buf, int sq, int sz, unsigned char*
data);

// parses control packet
int readControlPacket(unsigned char* buf, char* filename, int* filesz);

// parses data packet
int readDataPacket(unsigned char* data, unsigned char* buf, int* seq);
```

utils.c

```c
#include "utils.h"

int get_file_size(FILE* f) {
    int s;
    fseek(f, 0, SEEK_END);
    s = (int) ftell(f);
    fseek(f, 0, SEEK_SET);
    return s;
}

int numOfBytes(int filesz) {
    int counter = 0;
    while(filesz > 0) {
        filesz /= 256;
        counter++;
    }
    return counter;
}
```

```
int mountCtrlPacket(unsigned char* buf, int type, char* filename, int
filesz) {
    if(type == 0) {
        buf[0] = 0x02;
    }
    else if(type == 1) {
        buf[0] = 0x03;
    }

    unsigned L1 = numOfBytes(filesz);
    unsigned L2 = strlen(filename);
    unsigned ctrlsz = 5 + L1 + L2;

    buf[1] = 0x00;
    buf[2] = L1;

    for(int i = 0; i < L1; i++) {
        int tmp = (filesz & 0x0000FFFF) >> 8;
        buf[3 + i] = tmp;
        filesz = filesz << 8;
    }

    // memcpy(&buf[3],&filesz, L1);
    int nxt = L1 + 3;
    buf[nxt] = 0x01;
    buf[nxt + 1] = L2;
    memcpy(&buf[nxt + 2],filename, L2);

    /*for(int i = 4; i < 4 + ctrlsz; i++) {
        fprintf(stderr,"0x%02X, ", buf[i]);
    }*/

    return ctrlsz;
}

int mountDataPacket(unsigned char* buf, int sq, int sz, unsigned char*
data) {
    buf[0] = 0x01;
    buf[1] = (sq % 255);
    buf[2] = (sz / 256);
    buf[3] = (sz % 256);

    memcpy(buf + 4,data,sz);
    /*for(int i = 4; i < 4 + sz; i++) {
        fprintf(stderr,"0x%02X, ", buf[i]);
```

```
    }*/
    printf("\n\n");
    return (4 + sz);
}

int readControlPacket(unsigned char* buf, char* filename, int* filesz) {
    *filesz = 0;
    int fst;
    if(buf[0] == 0x02 && buf[0] == 0x03) {
        fprintf(stderr,"not ctrl packet");
        return -1;
    }

    if(buf[1] == 0x00) {
        fst = buf[2];
        for(int i = 0; i < fst; i++) {
            *filesz = *filesz * 256 + buf[3+i];
        }
    }
    else {
        return -1;
    }
    int snd = 0;
    int nxt = 5 + fst;

    if(buf[nxt - 2] == 0x01) {
        snd = buf[nxt - 1];
        for(int j = 0; j < snd; j++) {
            filename[j] = buf[j+nxt];
        }
    }
    else {
        return -1;
    }
    return 0;
}

int readDataPacket(unsigned char* data, unsigned char* buf, int* seq) {
    if(buf[0] != 0x01) {
        fprintf(stderr,"Not Data");
        return -1;
    }
    *seq = buf[1];
    int l2 = buf[2];
    int l1 = buf[3];
    int sz = (256 * l2) + l1;
```

```
    for(int i = 0; i < sz; i++) {
        data[i] = buf[4+i];
    }
    return sz;
}
```

state_machine.h

```
#include "macros.h"

enum stateMachine {START, FLAG_RCV, A_RCV, C_RCV, BCC_OK, C_INF, STOP,
REJ};

// changes the state depending on its current state and the byte it
receives
void changeState(enum stateMachine* st, unsigned char byte, int type);
```

state_machine.c

```
#include "state_machine.h"
#include <stdio.h>



void changeState(enum stateMachine* st, unsigned char byte, int type) {
    unsigned char store[2];
    if(type == 0) {
        switch (*st)
        {
        case START:
            if(byte == FLAG) {
                *st = FLAG_RCV;
            }
            else {
                *st = START;
            }
            break;

        case FLAG_RCV:
            if(byte == TRANS || byte == RECEIVE) {
                store[0] = byte;
                *st = A_RCV;
            }
            else if(byte == FLAG) {
                *st = FLAG_RCV;
```

```
            }
            else {
                *st = START;
            }
            break;

        case A_RCV:
            if((byte == UA) || (byte == SET) || (byte == DISC) || (byte
== 0x85) || (byte == 0x05)) {
                store[1] = byte;
                *st = C_RCV;
            }else if( (byte == 0x01) || (byte == 0x81) ){
                printf("BAD WRITE\n");
                store[1] = byte;
                *st = REJ;
            }
            else if(byte == FLAG) {
                *st = FLAG_RCV;
            }
            else if((byte == ICTRL_OFF) || (byte == ICTRL_ON)) {
                store[1] = byte;
                *st = C_INF;
            }
            else {
                *st = START;
            }
            break;

        case C_RCV:
            if(byte == BCC(store[0],store[1])) { //se  A^C = BBC ?
                *st = BCC_OK;
            }
            else if(byte == FLAG) {
                *st = FLAG_RCV;
            }
            else {
                *st = START;
            }
            break;
        case REJ:
            break;
        case BCC_OK:
            if(byte == FLAG) {
                *st = STOP;
            }
            else {
```

```
                *st = START;
            }
            break;

        case STOP:
            break;

        default:
            break;
        }
    }
    else if(type == 1) { // information frame
        switch (*st)
        {
        case START:
            if(byte == FLAG) {
                *st = FLAG_RCV;
            }
            else {
                *st = START;
            }
            break;

        case FLAG_RCV:
            if(byte == TRANS || byte == RECEIVE) {
                store[0] = byte;
                *st = A_RCV;
            }
            else if(byte == FLAG) {
                *st = FLAG_RCV;
            }
            else {
                *st = START;
            }
            break;

        case A_RCV:
            if(byte == FLAG) {
                *st = FLAG_RCV;
            }
            else if((byte == ICTRL_OFF) || (byte == ICTRL_ON)) {
                store[1] = byte;
                *st = C_INF;
            }
            else {
                *st = START;
```

```
            }
            break;

        case C_INF:
            if(byte == BCC(store[0],store[1])) { //se  A^C = BBC ?
                *st = BCC_OK;
            }
            else if(byte == FLAG) {
                *st = FLAG_RCV;
            }
            else {
                *st = START;
            }
            break;

        case BCC_OK:
            if(byte == FLAG) {
                *st = STOP;
            }
            break;

        case STOP:
            break;

        default:
            break;
        }
    }
}
```

macros.h

```
#define FLAG 0x7E
#define TRANS 0X03
#define RECEIVE 0X01
#define ESC 0x7D
#define ESCD 0x5D
#define ESCE 0x5E
#define SET 0x03
#define DISC 0x0B
#define UA 0x07
#define RR_0 0x05
#define RR_1 0x85
#define REJ_0 0x01
#define REJ_1 0x81
#define BCC(a,c) (a ^ c)
```

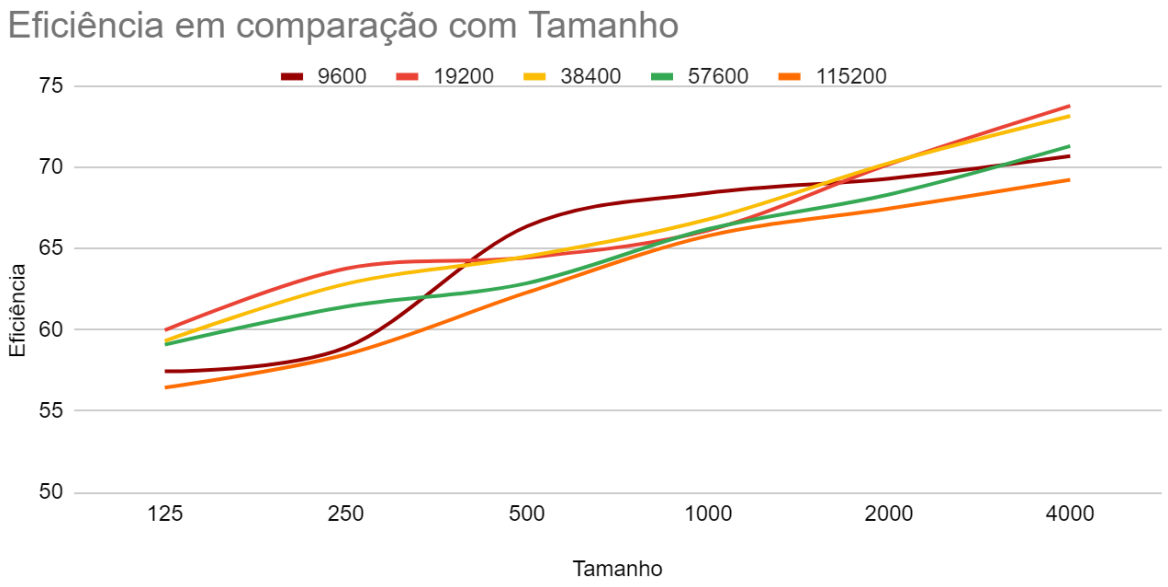
```
#define MAX_TRIES 3
#define BUF_SIZE 256
#define ICTRL_ON 0x40
#define ICTRL_OFF 0x00
```

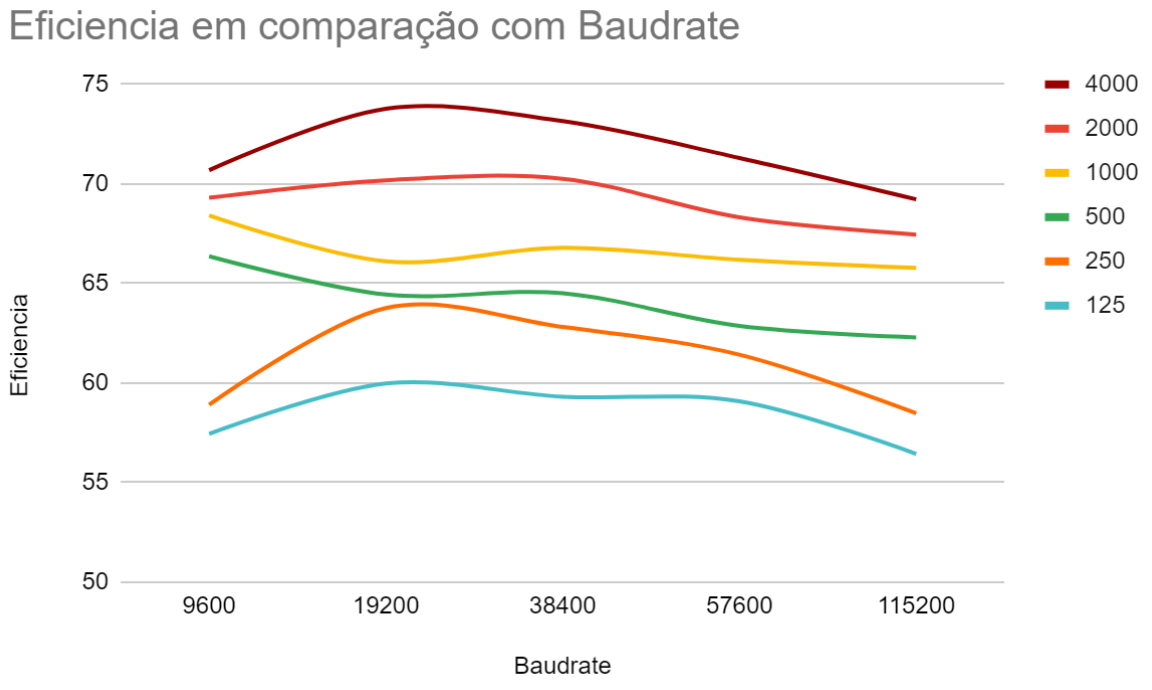
Anexo II - Testes

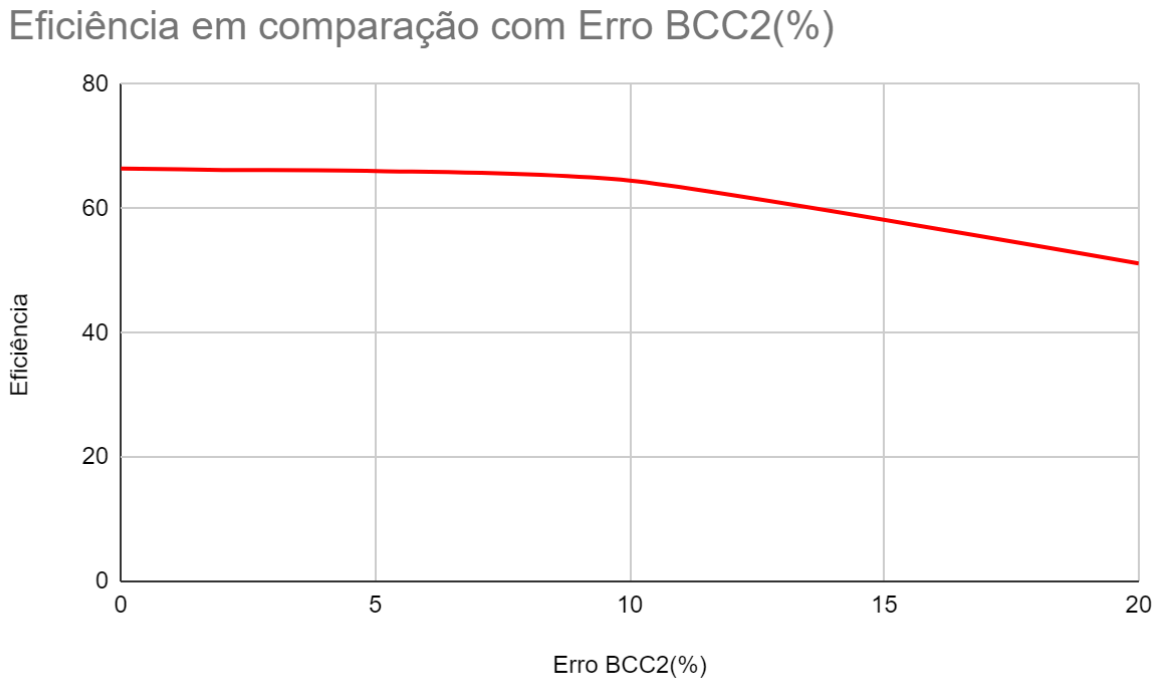
Medições:

Tamanho	Tempo	Baudrate	Velocidade	Eficiência
4000	12,9291	9600	6786,551268	70,69324238
4000	6,1939	19200	14166,19577	73,78226965
4000	3,1234	38400	28092,46334	73,15745662
4000	2,1363	57600	41072,88302	71,30708858
4000	1,1002	115200	79752,77222	69,229837
2000	13,1876	9600	6653,523006	69,30753132
2000	6,5114	19200	13475,44307	70,18459932
2000	3,2521	38400	26980,72015	70,26229206
2000	2,2294	57600	39357,67471	68,32929637
2000	1,1291	115200	77711,4516	67,45785729
1000	13,3581	9600	6568,598828	68,42290445
1000	6,9123	19200	12693,89349	66,11402862
1000	3,421	38400	25648,64075	66,79333528
1000	2,3012	57600	38129,67148	66,19734631
1000	1,1578	115200	75785,10969	65,7856855
500	13,7715	9600	6371,419235	66,36895037
500	7,0912	19200	12373,64621	64,44607401
500	3,5419	38400	24773,14436	64,51339676
500	2,4231	57600	36211,46465	62,86712613
500	1,2227	115200	71762,49284	62,29383059
250	15,5152	9600	5655,357327	58,90997216
250	7,1669	19200	12242,95023	63,76536578
250	3,6373	38400	24123,38823	62,82132351
250	2,4799	57600	35382,07186	61,42720809
250	1,3024	115200	67371,00737	58,48177723
125	15,9112	9600	5514,606064	57,44381316
125	7,6191	19200	11516,32083	59,98083763
125	3,8522	38400	22777,63356	59,31675406
125	2,5782	57600	34033,04631	59,08514985
125	1,3494	115200	65024,45531	56,44483968

Tamanho	Tempo	Baudrate	Erro BCC2(%)	Velocidade	Eficiência
500	17,8618	9600	20	4912,382851	51,17065469
500	14,172	9600	10	6191,363251	64,4933672
500	13,8412	9600	5	6339,33474	66,03473687
500	13,8031	9600	2	6356,832885	66,21700922
500	13,7549	9600	0	6379,108536	66,44904725






Eficiência em comparação com Erro BCC2(%)
80
60
40
20
0
Eficiência
0
5
10
15
20
Erro BCC2(%)