Ligação de dados

Relatório

RCOM - LEIC

João Gigante (up202008133@up.pt)

Ricardo Cavalheiro (up202005103@up.pt)

Sumário

Neste relatório consta o primeiro trabalho prático desenvolvido desde o começo do semestre. Este consiste no envio de ficheiros, entre computadores, por uma porta série assíncrona.

O projeto foi apresentado com sucesso, uma vez que a demonstração passou por todos os testes sem falhas. Confirmou-se assim a integridade do protocolo em assegurar a transferência de dados, mesmo que ocorram distúrbios durante a mesma.

Introdução

O objetivo do trabalho realizado foi implementar um protocolo de ligação de dados com o objetivo testar a realização da transferência de um ficheiro, em modo não canónico, através da porta série RS-232 que liga os dois sistemas.

Algumas características do protocolo são a existência de três tipos de tramas: Informação, Supervisão e Não-Numeradas, em que estas são delimitadas por flags e onde é utilizado o mecanismo de byte stuffing para evitar o falso reconhecimento de uma flag no interior de uma trama garantindo assim a transparência. É utilizada também a técnica de Stop and Wait de modo a resistir a alguns tipos de perturbação, como desativar a porta série ou até mesmo a presença de ruído.

Deste modo, o relatório está organizado da seguinte forma:

- **Arquitetura:** secção sobre os blocos funcionais e interfaces;
- **Estrutura do código:** demonstração das APIs, principais estruturas de dados utilizadas, principais funções e a sua relação com a arquitetura;
- **Casos de uso principais:** identificação dos casos de uso principais e sequência de chamada de funções;
- **Protocolo de ligação lógica:** identificação dos principais aspetos funcionais e descrição da estratégia implementada;
- **Protocolo de aplicação:** identificação dos principais aspetos funcionais e descrição da estratégia utilizada nestes aspetos;
- **Validação:** descrição dos testes efetuados com apresentação dos resultados;
- **Eficiência do protocolo de ligação de dados:** caracterização estatística da eficiência do protocolo efetuada recorrendo a medidas sobre o código desenvolvido;
- **Conclusões:** síntese da informação apresentada nas secções anteriores e reflexão sobre os objetivos de aprendizagem;
- **Anexos: código-fonte** e gráficos

Arquitetura

O trabalho está organizado em duas camadas bem distintas para assegurar a independência entre camadas.

A **camada de ligação de dados** é a responsável pelo estabelecimento da ligação, ou seja, todos os aspetos relativos à porta série, tratamento de erros, assim como o *byte stuffing* dos pacotes. A nível desta camada não existe nenhuma distinção entre pacotes de controlo e de dados nem é feito qualquer processamento que incida sobre o cabeçalho dos pacotes a transportar em tramas de informação, pois esta informação é inacessível ao protocolo de ligação de dados.

A **camada de aplicação** é a camada lógica situada acima da camada de ligação de dados, sendo a responsável pela transferência dos dados, permitindo a receção e emissão de tramas.

Estas duas camadas estão diretamente relacionadas, assim sendo a camada de aplicação depende diretamente da camada de ligação de dados.

Estrutura do código

1. Estruturas de dados
    a. LinkLayer

```
typedef struct
{
    char serialPort[50];
    LinkLayerRole role;
    int baudRate;
    int nRetransmissions;
    int timeout;
} LinkLayer;

enum STATE
{
    START,    // 0
    STOP,     // 1
    FLAG_RCV, // 2
    A_RCV,    // 3
    C_RCV,    // 4
    BCC_OK,   // 5
    IGNORE,   // 6
    REJECTED  // 7
};
```

2. Principais funções

```
void applicationLayer(const char *serialPort, const char *role,
int baudRate, int nTries, int timeout, const char *filename)
```

- Abre ficheiro
- Pede ao protocolo de ligação de dados para abrir uma conexão
  - Transmissor:
    - Divide ficheiro em segmentos e envia-os para o protocolo de ligação de dados
  - recetor:
    - Lê pacotes do protocolo de ligação de dados e escreve-os para um novo ficheiro
- Pede ao protocolo de ligação de dados para fechar a conexão

```
int llopen(LinkLayer connectionParameters)
```

- Abre a porta série e estabelece a conexão entre recetor e transmissor
- Guarda o valor do descritor de ficheiro para a porta série
- Retorna 1 em caso de sucesso

```
int llwrite(const unsigned char *buf, int bufSize)
```

- Escreve o pacote enviado através dos argumentos para o descritor de ficheiro
- Retorna número de bytes escrito

```
int llread(unsigned char *packet)
```

- Lê um pacote do descritor de ficheiro e guarda-o na variável passada através dos argumentos.
- Retorna o número de bytes lidos.

```
int llclose(int showStatistics)
```

- Fecha a conexão entre recetor e transmissor
- Fecha a porta série
- Restaura as definições da porta série

```
enum STATE next_state(enum STATE state, unsigned char byte,
unsigned char control, unsigned char command)
```

- Avançar a máquina de estados para o próximo estado se cumprir as condições
- Retorna o estado sucedido

Casos de uso principais

O trabalho é baseado em dois sistemas, um que atuará como emissor enviando o ficheiro escolhido e o outro que atuará como recetor do ficheiro enviado.

1. **Emissor**

Em relação ao emissor, após ser estabelecida a ligação, este irá iniciar a leitura e envio do ficheiro através de pacotes seguindo o protocolo de Stop and Wait. A seguir é feita a terminação da ligação.

- Abertura da porta série
- llopen_tx() : envio da trama SET, seguido pela recessão da trama UA do recetor;
- llwrite() : o pacote é montado e enviado para o recetor após ser feito o byte stuffing, conforme a trama que tenha recebido anteriormente; caso não tenha recebido nenhuma trama ou tenha recebido a trama REJ é realizada a retransmissão da trama de informação;
- llclose_tx() : envio da trama DISC, seguido pela receção da respetiva trama DISC do recetor e por fim o envio da trama UA;
- terminação da porta série;

2. **Recetor**

Em relação ao recetor, é feito o estabelecimento da ligação e,em seguida, inicia a receção dos pacotes provenientes do emissor, enviando as tramas de supervisão REJ ou RR como resposta, consoante a existência de erros na trama.

- Abertura da porta série;
- llopen_rx() : envio da trama UA após receção da trama SET do emissor;
- llread() : os pacotes de informação são recebidos e desmontados para verificar a existência de erros no cabeçalho, nos dados ou no número de sequência; caso existam é enviada a trama de supervisão REJ ou caso não existam a trama RR;
- llclose_rx() : receção da trama DISC do emissor, seguido pelo envio da trama DISC e, por fim, receção da trama UA;
- terminação da porta série;

Protocolo de ligação lógica

A camada de ligação de dados é a camada de mais baixo nível e é esta que comunica diretamente com a Porta Série.

O protocolo de ligação lógica permite:

- Abrir e encerrar a ligação da porta série
- Controlo dos erros de transmissão
- Disponibilização de uma API para comunicação através da porta série à camada de aplicação

Quanto à API foram implementadas as, quatros funções previamente planeadas: **llopen**, **llclose**, **llwrite**, **llread.**

Antes de falarmos das funções temos que falar da máquina de estados necessária para processar as mensagens recebidas. Esta processa cada byte lido e em caso de mensagem corretamente correspondida ela acaba no estado **STOP.**

A máquina tem 8 estados, START, STOP, FLAG_RCV, A_RCV, C_RCV, BCC_OK, IGNORE, REJECTED.

1. START é o estado inicial
2. STOP é o estado final
3. FLAG_RCV é o estado após a receção do byte **FLAG**
4. A_RCV é o estado após a receção do byte do campo de endereço
5. C_RCV é o estado após receção do byte de controle
6. BCC_OK é o estado após receção do byte **BCC1** e **confirmação** de este mesmo após uma operação xor entre os bytes guardados de controlo e de endereço.
7. IGNORE é o estado após bytes lidos não corresponderem com os pretendidos
8. REJECTED é o estado após byte recebido ser byte **REJ0** ou **REJ1**

A função **llopen** é responsável por estabelecer a ligação.

Começa por chamar a função **open**, abrindo a porta de série. Configura-se esta através da estrutura termios, com **VTIME a 0,1** e **VMIN a 0**, para que a função read não esteja à espera de um caractere antes de retornar.

Consoante o *role* passado como argumento, a função invocará o open correspondente, **llopen_rx** no caso do recetor, **llopen_tx** no caso do transmissor.

A função **llopen_tx** envia a mensagem **SET** e espera por uma resposta **UA.** Para confirmar a receção da resposta recorre à máquina de estados, que a cada byte recebido é chamada a função **next_state** que o processa e avança para o estado seguinte. Quando a máquina de estados avança para o estado **STOP** a função retorna sucesso.

A função **llopen_rx** espera pela mensagem de **SET** recorrendo novamente à máquina de estados. Todos os processos de leitura são semelhantes, todos usam a máquina de estados. Após a receção da mensagem, monta a resposta **UA** e envia-a.

A função **llwrite** é responsável por montar e enviar pacotes de informação através da porta série.

Começa por entrar num loop que a cada **timeout** ou resposta **REJ**, retransmite o pacote em caso de não receber a resposta **UA.**

Dentro do loop está também a lógica da construção do pacote. Começa por construir o **cabeçalho**, insere os **dados** passados no argumento da função e aplica-lhes o stuffing em caso do byte coincidir com o byte de **FLAG** ou **ESCAPE**, e por fim calcula o campo de verificação, o **BCC2,** que com o byte de **FLAG** fazem a **cauda** do pacote.

Se o estado no final do loop for o de **STOP,** atualiza os valores de **Ns** e **Nr** e retorna o número de bytes escritos, caso contrário retorna -1.

A função **llread** é responsável por ler e desmontar os pacotes de informação recebidos da porta série.

A lógica inicia-se por um primeiro loop que lê o **cabeçalho** do pacote. Se tudo estiver **OK** com este, dá-se início à leitura dos dados num outro loop, caso contrário este o pacote é ignorado.

A seguir lemos os dados byte a byte. Aplicamos o processo contrário ao **stuffing dos bytes** se este for igual ao byte de **ESCAPE** e a seguir vamos calcular o segundo campo de verificação, o **BCC2**. Terminamos o loop quando lemos uma **FLAG.**

Após o loop confirmamos que o **BCC2** calculado iguala o lido, através da aplicação do operador **xor** entre os dois e em caso do resultado ser 0, confirmamos resultado positivo. Em caso negativo, enviamos resposta **REJ**, em outro caso enviamos a resposta **RR** e retornamos o número de bytes lido.

A função **llclose** é responsável pelo encerramento da conexão.

De forma semelhante ao llclose é composto por um *switch* que o redireciona para uma de duas funções:

llclose_tx começa por montar o comando **DISC** e reenvia-o a cada **timeout** segundos em caso de não receber outro comando **DISC** do recetor. Após a receção do comando enviado pelo recetor, este monta a resposta **UA** e finaliza o protocolo

ou

llclose_rx que espera pela receção de um comando **DISC.** Após confirmada a receção do comando através do processo de leitura de byte a byte e recorrendo à máquina de estados, entra num loop de escrita. Envia o comando **DISC,** num máximo de número de retransmissões máximas, a cada **timeout** segundos em caso de não receber uma resposta **UA.**

A função **llclose** acaba, restaurando as definições originais da porta série e fecha o descritor de ficheiro da mesma. Retorna 1 em caso de sucesso e fecha o protocolo de ligação lógica.

Protocolo de aplicação

O protocolo de aplicação permite:

- Envio, receção e construção dos pacotes de controlo de início e fim de transmissão
- Leitura e escrita de dados por uma porta série utilizando o protocolo previamente desenvolvido
- Divisão de um ficheiro em fragmentos para envio posterior.

É na função **aplicationLayer** que desenvolvemos estes objetivos todos, usando recursos a algumas funções auxiliares. Começamos por **abrir os ficheiros**, tanto de entrada como de saída, a seguir a lógica divide-se tendo em conta o *role*.

No caso do transmissor, montamos o pacote de controle inicial, através da função **mount_control_packet,** e escrevemos este mesmo para a porta série usando o recurso do protocolo de ligação lógica, **llwrite**. Adiante procedemos à leitura do ficheiro por fragmentos de tamanho **MAX_PAYLOAD_SIZE** e respetiva construção do seu pacote de dados com header válido, com **mount_data_packet.** Por fim, enviamos o pacote de controle final para sinalizar o final da leitura e envio do ficheiro.

No caso do recetor, a lógica vai-se basear na leitura de pacotes. Todas as leituras recorrem ao protocolo de ligação lógica, **llread**. Começando pelo **pacote de controle** para dar início à construção do ficheiro e escrita. Dentro de um ***loop* lemos um pacote a cada iteração** e **escrevemos** no ficheiro de saída, após uma limpeza do header, até que este pacote se assemelhe a um **pacote de final de transmissão**, nesse caso o *loop* é terminado.

Procedemos então ao término da conexão nos dois lados para finalizar com o protocolo de aplicação.

Validação

Para testar a funcionalidade do nosso código tentamos correr os seguintes testes e todos foram completados com sucesso.

1. Transmissão do ficheiro pinguim.gif;
2. Transmissão de um ficheiro diferente do dado;
3. Desligar e ligar a porta série durante a transmissão;
4. Causar ruído na porta série durante a transmissão;
5. Variar os valores do **Baudrate**;
6. Variar os valores do tamanho da trama, do **MAX_PAYLOAD_SIZE;**
7. Envio de ficheiros com diferentes percentagens de erros simulados.

O sucesso destes permitiu-nos concluir que o protocolo e a aplicação funcionam como expectativa.

Eficiência do protocolo de ligação de dados

O protocolo implementado baseia-se num sistema Stop-and-Wait ARQ. O seu nome advém da natureza do comportamento. O emissor espera pela resposta do recetor a cada comando ou informação e o recetor envia automática uma retransmissão caso haja erros ou não corresponder com o esperado.

Todos os gráficos e dados estão apresentados no **Anexo II**.

1. Variar valores de **Baudrate**

Após a análise dos resultados obtidos e do gráfico podemos verificar uma relação inversamente proporcional entre o **Baudrate** e o Tempo Total de transferência do ficheiro. Quanto menor o **Baudrate**, maior o tempo de transferência e vice. Notamos que a eficiência se mantém constante, em 77%, concluímos assim que para pacotes de 1000 bytes o nosso programa passa 77% do seu tempo de execução ocupado com a transferência destes pacotes e o resto do tempo na sua construção, escrita para ficheiro…

2. Variar valores do tamanho da trama

Após a análise dos resultados obtidos e do gráfico podemos verificar uma **relação logarítmica** entre o tamanho da trama e o Tempo Total de transferência. Temos um aumento exponencial quando variamos o tamanho da trama de **8 bytes para 16 bytes**, de 88,83 segundos para menos de metade, 38,10 segundos. Já para valores de tamanho da trama maiores a diferença não se torna tão diferente, como para **1024 bytes** temos um tempo de 13,02 segundos e para **2048 bytes** apenas melhore para 12,89 segundos.

3. Variar percentagem de erros simulados

Após análise dos resultados obtidos e do gráfico podemos verificar **relações lineares se isolarmos o aumento de um tipo de erro do outro**. Se olharmos apenas para o aumento do erro d**o BCC2 notamos um aumento mínimo e linear, de cerca de 0,3 segundos a cada 2 pontos percentuais**. O contrário acontece se analisarmos apenas os valores de erro do **BCC1, onde a cada erro efetuado é aumentado 4 segundos ao Tempo Total de transferência**. Concluímos que isto acontece devido aquando um erro no BCC1 a trama é ignorada e nenhuma ação é realizada, contudo, um erro no BCC2 ativa uma resposta **REJ** que ignora o tempo de timeout e pede uma retransmissão imediata.

Conclusões

A realização deste trabalho permitiu-nos obter um melhor conhecimento acerca de alguns conceitos importantes como o mecanismo de *Stop and Wait*, a técnica de byte stuffing, assim como vários outros e foi essencial para aprendermos mecanismos sobre como lidar com erros.

Adicionalmente foi também possível verificar os benefícios aliados à independência entre camadas, permitindo que ao alterar uma camada, as restantes não tenham de ser alteradas.

De forma sucinta, os objetivos do trabalho foram concluídos com sucesso, o que permitiu aos elementos do grupo um aprofundamento, tanto teórico como prático, e um melhor entendimento sobre a estruturação e funcionamento de um protocolo de comunicação utilizando a porta série.

Anexo I

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

```
// Application layer protocol implementation

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

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

void applicationLayer(const char *serialPort, const char *role, int baudRate,
                      int nTries, int timeout, const char *filename)
{
    LinkLayer connectionParameters;
    LinkLayerRole r = strcmp(role, "tx") == 0 ? LITx : (LIRx);

    // Construct connection parameters
    strcpy(connectionParameters.serialPort, serialPort);
    connectionParameters.role = r;
```

```
    connectionParameters.baudRate = baudRate;
    connectionParameters.nRetransmissions = nTries;
    connectionParameters.timeout = timeout;

    // Open serial port
    if (!llopen(connectionParameters))
    {
        printf("Error opening serial port\n");
        exit(-1);
    }

    FILE *file = fopen(filename, "r");
    FILE *output = fopen("penguin-received.gif", "w");

    // Send file
    if (r == LITx)
    {
        unsigned char buffer[MAX_PAYLOAD_SIZE + 1],
control_packet[MAX_PAYLOAD_SIZE + 1];
        int size = get_file_size(file);

        printf("\nSending START control packet\n");
        mount_control_packet(control_packet, 2, size, filename);

        if (llwrite(control_packet, 5 + nBytes_to_represent(size) + strlen(filename)) == -1)
        {
            printf("Error sending control packet\n");
            llclose(0);
            exit(-1);
        }

        printf("Sending file...\n");
        int n = 0, sz, bytes;
        while ((sz = fread(buffer, 1, MAX_PAYLOAD_SIZE - 4, file)) > 0)
        {
            printf("Mount Data Packet #%d\n", n);
            unsigned char data_packet[MAX_PAYLOAD_SIZE];

            mount_data_packet(data_packet, buffer, sizeof(buffer), n);

            while (1)
            {
                if ((bytes = llwrite(data_packet, sz + 4)) == -1)
                {
                    printf("Error sending data packet\n");
                    llclose(0);
                    exit(-1);
```

```
            }

            if (bytes > 0)
                break;
        }

        n++;
    }

    printf("Sending END control packet\n");
    mount_control_packet(control_packet, 3, size, filename);

    if (llwrite(control_packet, 5 + nBytes_to_represent(sz) + strlen(filename)) == -1)
    {
        printf("Error sending control packet\n");
        llclose(0);
        exit(-1);
    }
}
else if (r == LlRx)
{
    unsigned char control_packet[MAX_PAYLOAD_SIZE],
data_packet[MAX_PAYLOAD_SIZE];

    printf("Receiving START packet...\n");
    if (llread(control_packet) == -1)
    {
        printf("Error receiving control packet\n");
        llclose(0);
        exit(-1);
    }

    printf("\nReceiving file...\n");
    int bytes;
    while (1)
    {
        printf("Reading Data Packet ");
        if ((bytes = llread(data_packet)) == -1)
        {
            printf("Error receiving data packet\n");
            llclose(0);
            exit(-1);
        }
        if (data_packet[0] == 3)
            break;

        if (bytes > 0) fwrite(data_packet + 4, 1, bytes - 4, output);
```

```
        }
    }
    else
    {
        printf("Invalid role: %s\n", role);
        exit(-1);
    }

    printf("\nDisconnecting!\n");
    if (!llclose(0))
    {
        printf("Error closing serial port\n");
        exit(-1);
    }

    printf("Application layer protocol finished\n");
    exit(0);
}
```

link_layer.h

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

#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
```

```c
#define _POSIX_SOURCE 1 // POSIX compliant source

// MISC
#define FALSE 0
#define TRUE 1
#define BUF_SIZE 256

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

int llopen_tx();
int llopen_rx();

// 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);

int llclose_tx();
int llclose_rx();

#endif // _LINK_LAYER_H_
```

link_layer.c

```c
// Link layer protocol implementation

#include "link_layer.h"
#include "state_machine.h"

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

```c
#include <signal.h>

// MISC
struct termios oldtio;
struct termios newtio;

int fd;

int Nr = 1;
int Ns = 0;

const char *serialPort;
int nRetries;
int timeout;
LinkLayerRole role;

int alarmEnabled = FALSE;
int alarmCount = 0;

unsigned char sequence_n = 0xff;

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

////////////////////////////////////////////////
// LLOPEN
////////////////////////////////////////////////
int llopen(LinkLayer connectionParameters)
{

    // Set global variables
    serialPort = connectionParameters.serialPort;
    role = connectionParameters.role;
    nRetries = connectionParameters.nRetransmissions;
    timeout = connectionParameters.timeout;

    // Open serial port
    fd = open(serialPort, O_RDWR | O_NOCTTY);

    if (fd < 0)
        exit(-1);
```

```
    // Save current port settings
    if (tcgetattr(fd, &oldtio) == -1)
        exit(-1);

    memset(&newtio, 0, sizeof(newtio));
    newtio.c_cflag = connectionParameters.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)
        exit(-1);

    if (role == LITx)
    {
        if (!llopen_tx())
        {
            printf("Error opening connection\n");
            exit(-1);
        }
    }
    else
    {
        if (!llopen_rx())
        {
            printf("Error opening serial port\n");
            exit(-1);
        }
    }

    return 1;
}

int llopen_tx()
{
    // Mount SET

    unsigned char buf[BUF_SIZE + 1] = {FLAG, A_SENDER, SET, A_SENDER ^ SET,
FLAG};

    enum STATE state = START;

    while (alarmCount < nRetries && state != STOP)
    {
```

```
        if (alarmEnabled == FALSE)
        {
            (void)signal(SIGALRM, alarmHandler);

            write(fd, buf, SET_SIZE);
            printf("SET written\n");

            state = START;

            alarm(timeout); // Set alarm to be triggered in 3s
            alarmEnabled = TRUE;
        }

        // Read from serial port
        int bytes;
        if ((bytes = read(fd, buf, 1)) <= 0)
        {
            if (bytes < 0)
                exit(-1);
            continue;
        }

        if (state == IGNORE)
            state = START;

        // Process byte
        if ((state = next_state(state, *buf, A SENDER, UA)) == STOP)
        {
            printf("UA received\n");
            break;
        }
    }
    return (state == STOP) ? 1 : -1;
}

int llopen_rx()
{

    // Loop for input
    unsigned char buf[BUF_SIZE + 1]; // +1: Save space for the final '\0' char

    // Receive SET
    enum STATE state = START;
    printf("Waiting for SET...\n");
    while (1)
    {
        if (state == IGNORE)
```

```
            state = START;

        int bytes;
        // Returns after 1 chars have been input
        if ((bytes = read(fd, buf, 1)) == 0)
            continue;

        if (state != 0)
            printf("%d and byte received: %d\n", state, bytes);

        // Process byte
        if ((state = next_state(state, *buf, A_SENDER, SET)) == STOP)
        {
            printf("SET received\n");
            break;
        }
    }

    printf("Sending UA...\n");
    buf[0] = FLAG;
    buf[1] = A_SENDER;
    buf[2] = UA;
    buf[3] = buf[1] ^ buf[2];
    buf[4] = FLAG;

    write(fd, buf, UA_SIZE);
    printf("UA written\n");
    return 1;
}

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

    int bytes = 0;
    alarmCount = 0;
    alarmEnabled = FALSE;
    enum STATE state = START;

    // Build frame
    while (alarmCount < nRetries && state != STOP)
    {
        if (alarmEnabled == FALSE || state == REJECTED)
        {
```

```
if (state == REJECTED)
    alarmCount++;
(void)signal(SIGALRM, alarmHandler);

// Build frame
_buf[0] = FLAG;
_buf[1] = A_SENDER;
_buf[2] = (Ns << 7);
_buf[3] = _buf[1] ^ _buf[2];

// Copy data
int i, j = 0;
unsigned char bcc2 = 0;
for (i = 0; i < bufSize; i++)
{
    // calculate bcc2
    bcc2 ^= buf[i];

    // byte stuffing
    if (buf[i] == FLAG || buf[i] == ESCAPE)
    {
        _buf[4 + i + j] = ESCAPE;
        _buf[4 + i + j + 1] = 0x5f & buf[i];
        j++;
    }
    else
    {
        _buf[4 + i + j] = buf[i];
    }
}

if (bcc2 == FLAG || bcc2 == ESCAPE)
{
    _buf[4 + i + j] = ESCAPE;
    _buf[4 + i + j + 1] = 0x5f & bcc2;
    j++;
}
else
{
    _buf[4 + i + j] = bcc2;
}
_buf[5 + i + j] = FLAG;

// Write frame
bytes = write(fd, _buf, 6 + i + j);
if (bytes < 0)
    exit(-1);
```

```
            state = START;
            alarm(timeout);
            alarmEnabled = TRUE;
        }

        // Returns after 1 chars have been input
        if (read(fd, _buf, 1) == 0)
            continue;

        // Process byte
        state = next_state(state, *_buf, A_SENDER, (RR | (Nr << 7)));
        if (state == STOP)
        {
            printf("Response received\n");
            break;
        } else if (state == REJECTED)
        {
            printf("Response rejected\n");
            read(fd, _buf, 1);
            read(fd, _buf, 1);
        }
    }

    if (state == STOP)
    {
        Nr = Ns;
        Ns = (Ns + 1) % 2;
        return bytes;
    }
    else
    {
        printf("Error: llwrite failed\n");
        return -1;
    }
}

////////////////////////////////////////////////
// LLREAD
////////////////////////////////////////////////

int llread(unsigned char *packet)
{
    unsigned char buf[BUF_SIZE + 1]; // +1: Save space for the final '\0' char
    enum STATE state = START;

    // Receive packet
```

```
while (state != BCC_OK)
{
    // Returns after 1 chars have been input
    if (read(fd, buf, 1) == 0)
        continue;

    // Process byte
    if (state == A_RCV && (*buf == (0 << 7) || (*buf == (1 << 7))))
    {
        Ns = (*buf >> 7);
        Nr = (Ns + 1) % 2;
    }

    state = next_state(state, *buf, A_SENDER, Ns << 7);
    if (state == IGNORE)
        return 0;
}

// Read data
unsigned char bcc2 = 0;
int i = 0, data = 0;
while (state != STOP)
{
    if (state == IGNORE || state == REJECTED)
        break;

    // Returns after 1 chars have been input
    if (read(fd, buf, 1) == 0)
        continue;

    // If data packet
    if (i == 0)
        data = (*buf == 1);

    // Record sequence number
    if (i == 1 && data)
    {
        printf("#%d \n", *buf);
        if (sequence_n == *buf)
        {
            state = IGNORE;
            printf("Repeated packet\n");
            break;
        }
        sequence_n = *buf;
    }
```

```
    if (*buf == FLAG)
    {
        state = STOP;
        break;
    }

    if (*buf == ESCAPE)
    {
        // byte destuffing
        while (*buf == 0x7d)
            read(fd, buf, 1); // read next byte

        if (*buf == 0x5e)
        {
            *buf = FLAG;
        }
        else if (*buf == 0x5d)
        {
            *buf = ESCAPE;
        }
    }

    // Copy data
    *(packet + i) = *buf;
    i++;
    bcc2 ^= *buf;
}

if (bcc2 != 0)
{
    printf("BCC2 not ok\n");
    state = REJECTED;
    sequence_n = sequence_n - 1;
}

*(packet + i) = '\0';
i--;

// Send RR or REJ
buf[0] = FLAG;
buf[1] = A_SENDER;
buf[2] = (state != REJECTED ? RR : REJ) | (Nr << 7);
buf[3] = buf[1] ^ buf[2];
buf[4] = FLAG;

int bytes = write(fd, buf, RR_SIZE);
printf("Wrote %s\n", state != REJECTED ? "RR" : "REJ");
```

```
    if (state == REJECTED)
        return 0;
    return i;
}

////////////////////////////////////////////////
// LLCLOSE
////////////////////////////////////////////////
int llclose(int showStatistics)
{

    // Close serial port
    if (role == LITx)
    {
        if (!llclose_tx())
        {
            printf("Error closing serial port\n");
            exit(-1);
        }
    }
    else
    {
        if (!llclose_rx())
        {
            printf("Error closing serial port\n");
            exit(-1);
        }
    }

    // Restore the old port settings
    if (tcsetattr(fd, TCSANOW, &oldtio) == -1)
    {
        perror("tcsetattr");
        exit(-1);
    }

    close(fd);

    return 1;
}

int llclose_tx()
{

    // Mount DISC
    alarmEnabled = FALSE;
```

```
    alarmCount = 0;

    unsigned char buf[BUF_SIZE + 1] = {FLAG, A_SENDER, DISC, A_SENDER ^ DISC,
FLAG}, _buf[BUF_SIZE];

    enum STATE state = START;
    while (alarmCount < nRetries && state != STOP)
    {
        if (alarmEnabled == FALSE)
        {
            (void)signal(SIGALRM, alarmHandler);

            write(fd, buf, DISC_SIZE);
            printf("DISC written from tx\n");

            state = START;

            alarm(timeout); // Set alarm to be triggered in timeout (s)
            alarmEnabled = TRUE;
        }

        if (state == IGNORE)
            state = START;

        // Read from serial port
        // Returns after 1 chars have been input
        int bytes = read(fd, _buf, 1);
        if (bytes < 0)
            exit(-1);
        else if (bytes == 0)
            continue;

        if ((state = next_state(state, *_buf, A_RECEIVER, DISC)) == STOP)
        {
            printf("DISC received\n");
        }
    }

    if (state != STOP)
    {
        printf("Error: llclose failed\n");
        return -1;
    }

    // Send UA
    buf[0] = FLAG;
    buf[1] = A_RECEIVER;
```

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

    if (!write(fd, buf, UA_SIZE))
    {
        printf("Error sending UA\n");
        exit(-1);
    }

    printf("Sent UA\n");

    sleep(0.5);

    return (state == STOP) ? 1 : -1;
}

int llclose_rx()
{
    unsigned char buf[BUF_SIZE + 1] = {0};

    // Receive DISC
    enum STATE state = START;
    while (1)
    {
        // Returns after 1 chars have been input
        if (read(fd, buf, 1) == 0)
            continue;

        if (state == IGNORE)
            state = START;

        // Process byte
        if ((state = next_state(state, *buf, A_SENDER, DISC)) == STOP)
        {
            printf("DISC received\n");
            break;
        }
    }


    // Mount DISC
    unsigned char _buf[BUF_SIZE + 1] = {FLAG, A_RECEIVER, DISC, A_RECEIVER ^
DISC, FLAG, '\0'};
    // Set alarm
    alarmEnabled = FALSE;
    alarmCount = 0;
```

```
    state = START;

    while (alarmCount < nRetries && state != STOP)
    {

        if (alarmEnabled == FALSE)
        {
            (void)signal(SIGALRM, alarmHandler);

            write(fd, _buf, DISC_SIZE);
            printf("DISC written from rx\n");

            state = START;

            alarm(timeout); // Set alarm to be triggered in 3s
            alarmEnabled = TRUE;
        }

        if (state == IGNORE)
            state = START;

        // Read from serial port
        int bytes = read(fd, _buf, 1);
        if (bytes < 0)
            exit(-1);
        if (bytes == 0)
            continue;


        if ((state = next_state(state, *_buf, A_RECEIVER, UA)) == STOP)
        {
            printf("UA received\n");
            break;
        }
    }

    return (state == STOP) ? 1 : -1;
}
```

state_machine.h

```
#ifndef STATE_MACHINE_H
#define STATE_MACHINE_H

#define FLAG 0x7e       // 0111 1110
#define ESCAPE 0x7d     // 0111 1101
#define ESCAPE_MASK 0x20 // 0010 0000
```

```
#define SET_SIZE 6
#define UA_SIZE 6
#define DISC_SIZE 6
#define RR_SIZE 6
#define REJ_SIZE 6

#define A_SENDER 0x03   // 0000 0011
#define A_RECEIVER 0x01 // 0000 0001

#define SET 0x03  // 0000 0011
#define DISC 0x0b // 0000 1011

#define UA 0x07 // 0000 0111

#define TS_MASK 0x7f // 0111 1111

#define RR 0x05   // 0000 0101
#define RR0 0x05  // 0000 0101
#define RR1 0x85  // 1000 0101
#define REJ 0x01  // 0000 0001
#define REJ0 0x01 // 0000 0001
#define REJ1 0x81 // 1000 0001

#define TI_MASK 0xbf // 1011 1111

#define TI 0x00  // 0000 0000
#define TI0 0x00 // 0000 0000
#define TI1 0x40 // 0100 0000

#define MAX_SIZE 256

enum STATE
{
   START,    // 0
   STOP,     // 1
   FLAG_RCV, // 2
   A_RCV,    // 3
   C_RCV,    // 4
   BCC_OK,   // 6
   IGNORE,   // 7
   REJECTED  // 8
};

enum STATE next_state(enum STATE state, unsigned char byte, unsigned char control,
unsigned char command);
```

```
#endif // STATE_MACHINE_H
```

state_machine.c

```
#include "state_machine.h"

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

enum STATE next_state(enum STATE state, unsigned char byte, unsigned char control,
unsigned char command)
{
    switch (state)
    {
    case START:
        if (byte == FLAG)
        {
            return FLAG_RCV;
        }
        else
        {
            return START;
        }
        break;
    case FLAG_RCV:
        if (byte == control)
        {
            return A_RCV;
        }
        else if (byte == FLAG)
        {
            return FLAG_RCV;
        }
        else
        {
            return IGNORE;
        }
        break;
    case A_RCV:
        if (byte == command)
        {
            return C_RCV;
        }
        else if (byte == (REJ << (command >> 7))) {
            return REJECTED;
        }
        else if (byte == FLAG)
```

```
        {
            return FLAG_RCV;
        }
        else
        {
            return IGNORE;
        }
        break;
    case C_RCV:
        if (byte == (control ^ command))
        {
            return BCC_OK;
        }
        else if (byte == FLAG)
        {
            return FLAG_RCV;
        }
        else
        {
            return IGNORE;
        }
        break;
    case BCC_OK:
        if (byte == FLAG)
        {
            return STOP;
        }
        else
        {
            return IGNORE;
        }
        break;
    case STOP:
        return STOP;
        break;
    default:
        return START;
        break;
    }
}
```

utils.h

```
#ifndef UTILS_H
#define UTILS_H

#include <stdio.h>
```

```
#include <stdlib.h>

int get_file_size(FILE *filename);

int nBytes_to_represent(int n);

int mount_control_packet(unsigned char *control_packet, int start, int file_size, const char
*filename);

int mount_data_packet(unsigned char *data_packet, unsigned char *buffer, int size, int n);

#endif // UTILS_H
```

utils.c

```
#include "utils.h"

#include <string.h>

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

int nBytes_to_represent(int n)
{
    int i = 0;
    while (n > 0)
    {
        n = n / 256;
        i++;
    }
    return i;
}

int mount_control_packet(unsigned char *control_packet, int start, int file_size, const char
*filename)
{
    control_packet[0] = start; // START
    // FILE SIZE
    control_packet[1] = 0; // T1
    int l1 = nBytes_to_represent(file_size);
    control_packet[2] = l1;                    // L1
```

```
    strncpy(control_packet + 3, (const char *) &file_size, l1); // V1

    // FILE NAME
    control_packet[4 + (l1 - 1)] = 1;                          // T2
    control_packet[5 + (l1 - 1)] = strlen(filename);            // L2
    strncpy(control_packet + 6 + (l1 - 1), filename, strlen(filename)); // V2

    return 1;
}

int mount_data_packet(unsigned char *data_packet, unsigned char *buffer, int size, int n)
{
    data_packet[0] = 1;              // DATA
    data_packet[1] = n;              // N
    data_packet[2] = size / 256;     // L2
    data_packet[3] = size % 256;     // L1
    memcpy(data_packet + 4, buffer, size); // V2

    return 1;
}
```

Anexo II

Variar tamanho da trama	Tempo (s)	Velocidade (b/s)	Eficiência (Velocidade/Baudrate)
8	88,83	987,76	10,29%
8	88,84	987,62	10,29%
16	38,12	2 302,04	23,98%
16	38,10	2 303,04	23,99%
32	23,62	3 714,06	38,69%
32	23,63	3 713,64	38,68%
64	17,84	4 917,20	51,22%
64	17,83	4 920,34	51,25%
128	15,13	5 797,96	60,40%
128	15,16	5 788,03	60,29%
256	13,93	6 299,05	65,62%
256	13,93	6 298,48	65,61%
512	13,32	6 585,64	68,60%
512	13,32	6 587,11	68,62%
1024	13,03	6 734,10	70,15%
1024	13,02	6 736,72	70,17%
2048	12,89	6 805,64	70,89%
2048	12,89	6 807,06	70,91%

Fig 1. Dados para teste de variação do tamanho da trama

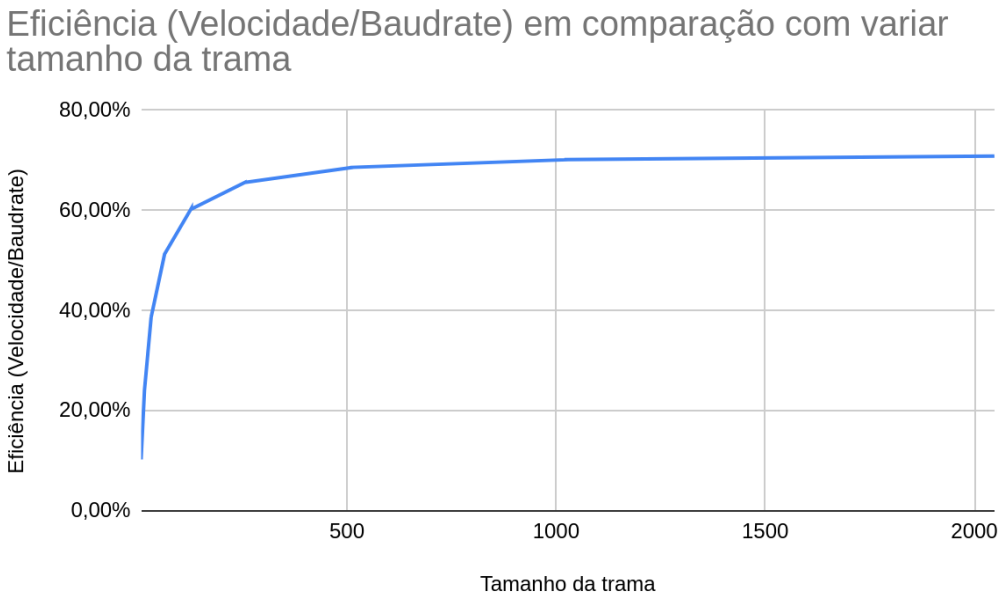

Fig 2. Gráfico da eficiência de acordo com a variação da trama

Variar Baudrate	Tempo (s)	Velocidade (b/s)	Eficiência (Velocidade/Baudrate)
4800	23,70	3 702,50	77,14%
4800	23,69	3 703,35	77,15%
9600	11,85	7 404,30	77,13%
9600	11,85	7 404,30	77,13%
19200	5,93	14 805,17	77,11%
19200	5,93	14 806,14	77,12%
38400	2,96	29 596,13	77,07%
38400	2,96	29 599,90	77,08%
57600	1,98	44 382,29	77,05%
57600	1,98	44 373,17	77,04%

Fig 3. Dados para teste de variação de **Baudrate**

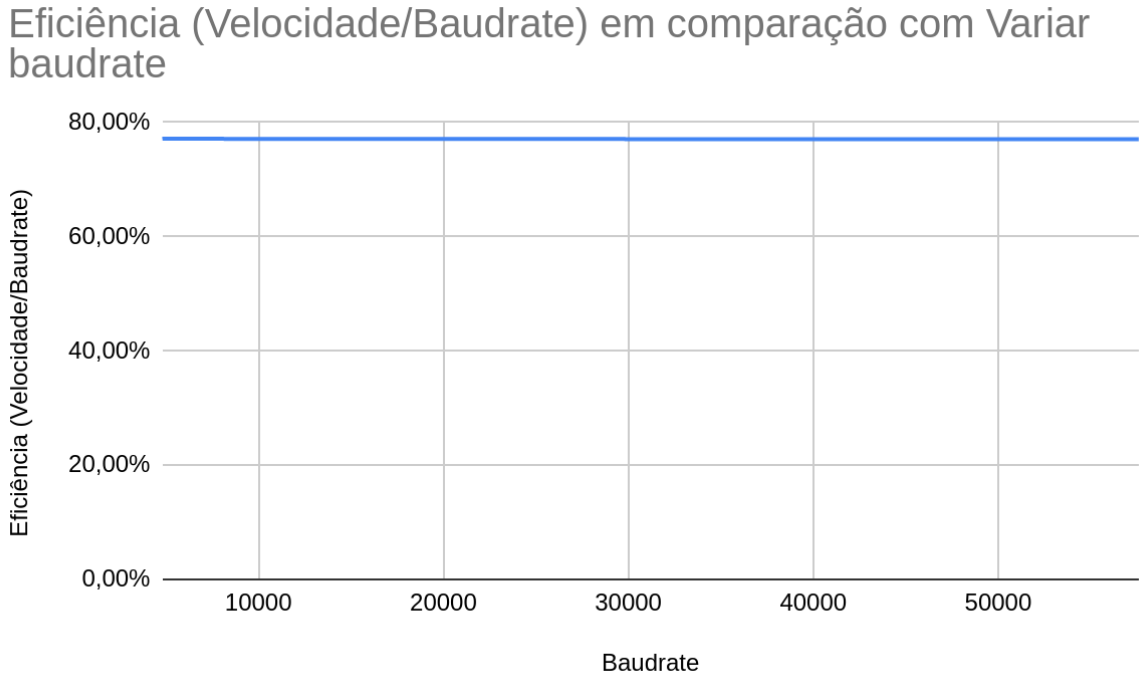

Fig 4. Gráfico da eficiência de acordo com a variação do **Baudrate**

Probabilidade do erro		Tempo (s)	Velocidade (b/s)	Eficiência (Velocidade/Baudate)
bcc1	bcc2			
0	0	6,47	13 563,16	70,64%
2	0	10,47	8 379,85	43,65%
0	2	6,70	13 103,31	68,25%
2	2	10,58	8 291,18	43,18%
4	2	14,69	5 971,14	31,10%
2	4	11,34	7 735,27	40,29%
4	4	15,35	5 715,15	29,77%

Fig 5. Dados para teste de variação de erros simulados

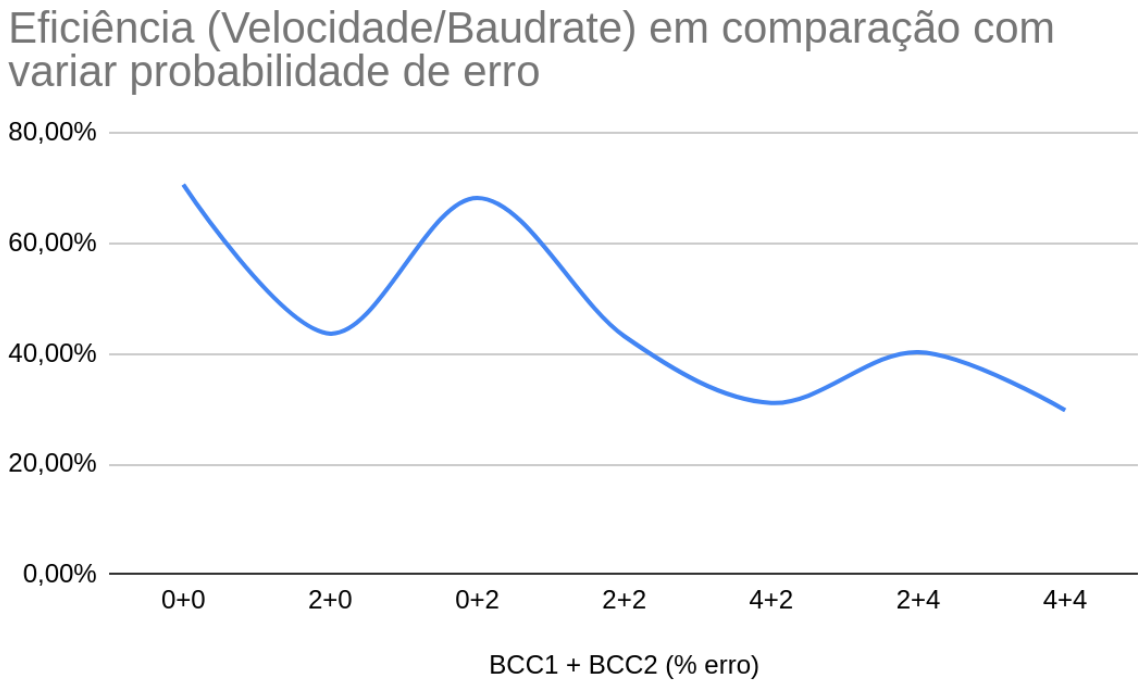

Fig 6. Gráfico da eficiência de acordo com a variação dos erros nos campos de validação