U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Redes de Computadores

1º Trabalho Laboratorial

6 de novembro de 2017

Carlos Freitas – up201504749
David Falcão – up201506571
Luís Martins – up201503344

Índice

Sumário

No âmbito do primeiro trabalho laboratorial de Redes de Computadores, este relatório tem como objetivo realizar uma análise estatística ao comportamento do protocolo implementado para a transmissão e receção de ficheiros através de uma porta de série assíncrona.

O trabalho foi realizado com sucesso, uma vez que todos os objetivos propostos foram atingidos, verificando-se que o protocolo é capaz de assegurar uma transmissão correta dos dados, mesmo que ocorram alterações nos pacotes durante a mesma, como por exemplo interrupções ou curtos-circuitos na ligação, alterando os dados.

Introdução

O objetivo do trabalho realizado era implementar um protocolo de ligação através da porta de série, utilizando a técnica *Stop and Wait*, de modo que fosse resistente a alguns fatores de perturbação, como interrupções e curtos-circuitos durante a sua execução. Para tal, foi utilizada a abstração do envio de um ficheiro de teste, para verificar a funcionalidade e resistência do protocolo implementado.

Para garantir coesão nos dados enviados e recebidos, foram utilizadas algumas técnicas de proteção, nomeadamente adição de bytes de controlo e técnica de transparência (*framming*), permitindo assim também fazer um controlo ao fluxo do programa.

O presente relatório tem como objetivo analisar e interpretar o resultado obtido em relação à implementação do protocolo, analisando os dados obtidos através de testes.

Assim, este relatório está organizado da seguinte forma:

- **Arquitetura e estrutura de código:** Demonstração dos blocos funcionais e interfaces. Principais estruturas de dados, principais funções e a sua relação com a arquitetura.
- **Casos de Uso Principais:** Identificação dos principais casos de uso e sequência da chamada de funções.
- **Protocolo de ligação lógica:** Identificação dos principais aspetos funcionais e descrição da estratégia de implementação destes.
- **Protocolo de aplicação:** Identificação dos principais aspetos funcionais e descrição da estratégia de implementação destes.
- **Validação:** Apresentação dos testes efetuados e apresentação quantificada dos resultados obtidos.
- **Eficiência do protocolo de ligação de dados:** Comparação do programa com mecanismos *Stop and Wait* de *ARQ* e apresentação da eficiência do mesmo através dos testes apresentados.
- **Conclusão:** Síntese do projeto realizado e reflexão sobre os principais objetivos de aprendizagem alcançados.

Arquitetura e Estrutura do código

O trabalho está organizado em duas principais camadas lógicas, que permitem uma correta organização e funcionamento do protocolo, tornando-o modular.

Camada de Ligação (Data Link Layer)

A camada de Ligação é a camada de software "mais baixa" e é responsável pela comunicação e verificação da coesão dos resultados provenientes da porta de série.

Deste modo, esta camada contém todas as funções relativas ao estabelecimento da ligação entre os terminais (*LLOPEN*), escrita e leitura da informação utilizando a porta série (*LLWRITE/LLREAD*), e respetiva análise em relação à sua coesão (*state_machine* e *verify_bcc2*). Esta camada é também responsável pelo fecho da ligação no fim da comunicação (*LLCLOSE*).

Para uma melhor estruturação, esta camada contém uma estrutura de dados (struct), por forma a manter algumas informações importantes para a execução das funções de leitura e escrita de e para a porta de série, nomeadamente:

- o tempo de timeout
- o número máximo de retransmissões
- o valor de controlo da última trama, para que se possa verificar se se trata de um duplicado

```
typedef struct {
   struct termios oldtio;
   struct termios newtio;
   unsigned char control_value;
   unsigned int timeout;
   unsigned int max_transmissions;
} link_layer;
```

Camada de Aplicação (Application Layer)

A camada de Aplicação é a camada "mais alta" e é responsável pela inicialização do programa (*main*). É a partir desta que se desenrola o processo de transferência de dados através da porta de série. Esta camada é também responsável pela criação dos pacotes que contêm os dados a enviar, tanto as tramas de START e END (*create_STARTEND_packet*) como as tramas de Informação (*data_package_constructor*).

Assim, esta camada contém, entre outras funções, as funções de leitura (*handle_readfile*) e de criação/escrita (*handle_writefile*) do ficheiro a transmitir, no caso de o programa ser executado no modo de escritor ou no modo de leitor, respetivamente.

Para uma melhor estruturação, esta camada contém duas estruturas de dados essenciais para guardar informação:

- a primeira relativa ao ficheiro de leitura/escrita (nome, tamanho, ...)
- a segunda relativa aos dados a enviar para a camada de ligação, que se referem à comunicação com a porta de série (descritor da porta)

```
typedef struct {
  int filesize;
  char* filename;
  FILE* fp;
  int size_to_read;
} file_info;

typedef struct {
  int file_descriptor;
  char* status;
}application_layer;
```

Casos de Usos Principais

A aplicação resultante possui vários casos de uso, nomeadamente os que são responsáveis pela transferência do ficheiro e pela inserção de dados por parte do utilizador. Assim, a transferência do ficheiro dá-se da seguinte forma:

No caso do emissor:

O emissor preenche a estrutura file com os dados do ficheiro necessários à abertura e leitura do mesmo. O restante funcionamento do programa divide-se em 3 partes fundamentais, e sequenciais:

- Criação e envio da trama START utilizando as funções *create_STARTEND_packet()* e *send_message()*, sendo que esta última função envia a trama para a camada de ligação através da chamada à função *LLWRITE*;
- Leitura e envio dos vários pacotes de dados provenientes do ficheiro. Para tal, é chamada a função *handle_readfile()* que lê o ficheiro e chama a função *send_message()* para envio da informação lida;
- Por fim, é criada e enviada, de forma idêntica à START, a trama END.

De referir que a função *send_message(),* no caso das tramas de Informação, invoca a função *data_package_constructor()* que adiciona o cabeçalho ao pacote, nomeadamente os valores do número do pacote e o tamanho de dados a enviar.

É também importante referir que a função *LLWRITE*, adiciona a restante informação necessária à trama através da chamada à função *create_frame(),* que por sua vez calcula o valor de BCC2 e invoca as funções *byte_stuffing()* e *add_frame_header(),* para que seja realizada a função de transparência e sejam adicionados os cabeçalhos. No fim, a função *LLWRITE* envia a mensagem através da chamada de sistema *read()*.

No caso do recetor:

É invocada a função *get_message()* que espera uma mensagem proveniente do emissor, através da chamada à função *LLREAD*. A função *LLREAD* lê e retorna o pacote recebido após verificar se existe a coesão dos dados recebidos (BCC1, BCC2, e duplicados). Posteriormente, esta trama é analisada e verifica-se de que tipo de trama se trata. Caso se trate de um pacote de informação, é

invocada a função *get_only_data()* que verifica de qual dos seguintes tipos de pacotes se trata, retornando o pacote lido:

- Mensagem START: chama a função *start_message(),* que preenche os dados da estrutura file com os respetivos dados para a criação do ficheiro, criando-o;
- Pacotes de Informação: chama a função *get_only_data(),* que remove o cabeçalho do pacote e posteriormente chama a função *handle_writefile(),* que escreve os dados para o ficheiro de destino;
- Mensagem END: chama a função *verify_end(),* que verifica se o tamanho do ficheiro recebido corresponde ao tamanho recebido na trama final;
- Se a trama recebida for um DISC ou nula, nada é analisado.

O recetor executa a função *get_message()* enquanto esta não retornar uma trama DISC, sinal de que a execução do programa terminou e a porta de série já foi fechada.

Em ambos os casos vai ser executada no início a função *LLOPEN,* para abrir e configurar a porta de série.

Quanto à interação com o utilizador, este apenas deve iniciar o programa executando através do terminal um comando com a seguinte estrutura:

- **Emissor:** {Nome da aplicação} {nome da porta de série (ex: /dev/ttyS0)} {modo de abertura (w)} *{nome do ficheiro a enviar}* {tempo de timeout} {número de tentativas máximas de reenvio}
- **Recetor:** {Nome da aplicação} {nome da porta de série (ex: /dev/ttyS0)} {modo de abertura (r)} {tempo de timeout} {número de tentativas máximas de reenvio}

Protocolo de Ligação Lógica

A camada de ligação é uma das camadas implementadas neste projeto. Esta camada é responsável pela comunicação e coesão da mesma através de verificações feitas aos valores BCC1, BCC2 e C1. Assim, esta camada tem as seguintes funcionalidades:

- Estabelecer e terminar uma ligação através da porta de série (Envio e receção das tramas SET e UA);
- Escrever e ler mensagens através da porta série (Envio e receção das tramas RR, REJ e I);
- Adicionar e remover os cabeçalhos necessários para o envio das tramas S e I referentes ao envio dos pacotes de informação, e respetivo *stuff/destuff*;
- Analisar os cabeçalhos das mensagens recebidas enviando a respetiva trama de resposta.

Deste modo, as principais funções desta camada são as seguintes:

LLOPEN

*__int__ LLOPEN(__char*__ port, __char*__ mode, __char*__ timeout, __char*__ max_transmissions) ;*

A função *llopen* é responsável por estabelecer a ligação através da porta de série.

Quando o emissor invoca esta função é feito o envio do comando SET e aguarda a respetiva resposta UA. Em caso de sucesso, a ligação fica estabelecida e a função retorna o valor do descritor da ligação através da porta de série. Caso contrário, existem as seguintes possibilidades:

- se UA não for recebido: a trama SET é reenviada passados timeout segundos até um máximo de max_transmissions tentativas. Caso seja atingido o número máximo de tentativas, a função retorna -1, indicando que a ligação não foi estabelecida.
- caso exista resposta por parte do recetor mas esta não seja afirmativa, o emissor reenvia a trama SET até que a resposta seja afirmativa.

Já quando o recetor invoca esta função, fica a aguardar a chegada de um comando SET e quando o receber, envia como resposta UA. Deste modo, a ligação fica estabelecida.

LLWRITE

*__int__ LLWRITE(__int__ fd, unsigned __char*__ msg, __int*__ length);*

A função *llwrite* é responsável pelo envio dos pacotes de Informação.

Assim esta função recebe como argumento, entre outros, a mensagem a ser enviada (pacotes de controlo + dados) e o respetivo tamanho. Deste modo, com recurso à função *create_package()* esta cria a trama com os cabeçalhos necessário (flags, campo de endereço, campo de controlo(c1), BCC1, BCC2 e respetivo stuffing) e envia a mesma através da porta de série.

O envio das tramas de informação é similar ao estabelecimento da ligação por parte do emissor, ou seja, a mensagem é enviada e fica-se a aguardar resposta sendo reenviada a mesma quando for atingido o tempo de timeout. Caso a resposta recebida seja o comando RR (C1 = 0x05/0x85), a mensagem foi recebida corretamente e pode-se enviar o próximo pacote; caso a resposta seja o comando REJ(C1 = 0x01/0x81), o último pacote é reenviado. A função retorna TRUE em caso de sucesso e FALSE caso contrário.

LLREAD

*__unsigned char*__ LLREAD(__int__ fd, __int*__ length);*

A função *llread* é responsável pela leitura dos pacotes de informação provenientes da porta de série. Como tal, esta função, ao ser invocada lê a informação, um char de cada vez, verificando o início e fim da trama(FLAG) e coesão do cabeçalho (BCC1), através da chamada a função *state_machine().* São também invocadas aqui as funções de introdução de erros aleatórios no cabeçalho e dados. Caso não ocorra nenhum erro no BCC1 a função prossegue.

Nesta mesma função analisa-se a trama recebida verificando os seguintes parâmetros

- se a trama recebida se trata de um DISC (Sinal de terminação de envio das tramas de informação), é invocado LLCLOSE para se proceder ao protocolo de fecho da porta de série;
- se não, é analisada a presença de um duplicado (verificando o valor do campo de controlo), é feito o *destuffing* e a respetiva verificação do valor de BCC2(*verify_bcc2()*)

Posteriormente é enviada, através da função *send_response(),* a resposta de acordo com os dados recebidos, segundo a seguinte lógica:

- Trata-se de um duplicado: é enviado RR;
- Não é duplicado e BCC2 está correto: é enviado RR;
- Não é duplicado e BCC2 está errado: é enviado REJ.

Caso seja enviado RR, será o valor correspondente a RR do complemento do tipo de trama recebida, ou seja, caso se receba uma trama 1 (C = 0x40) envia-se RR correspondente a RR0(0x05). Já no caso de ser enviado REJ, será o valor correspondente a REJ do tipo de trama recebida, ou seja, se se receber uma trama 1, envia-se REJ correspondente a REJ(0x81).

LLCLOSE

```
void LLCLOSE(int fd, int type);
```

A função LLCLOSE é responsável por terminar a ligação através da porta de série. Esta função tem comportamentos distintos caso se trate do recetor ou do emissor.

Assim, no caso do recetor, esta função é chamada na função LLREAD ao ser encontrado um comando DISC. Após isso, o recetor através da chamada à função *send_disc()* reenvia o comando DISC, aguardando uma resposta UA final.

Já no caso do emissor, esta função é chamada após o envio e receção da respetiva resposta da trama END. É chamada a função *send_disc()* que envia uma trama DISC e espera uma resposta idêntica.

Em caso de sucesso, a ligação é também terminada no lado do emissor.

Protocolo de Aplicação

A camada de aplicação é a camada de mais alto nível implementada neste projeto. Esta camada é responsável pelas seguintes funcionalidades:

- Envio/receção dos pacotes de dados e de controlo
- Leitura/escrita do ficheiro a enviar/recebido.

Uma vez que é nesta camada que se inicia todo o processo, a função *main* tem dois blocos e escolhe qual executar, dependendo se se trata do emissor ou do recetor, parâmetro este que é recebido como argumento da função.

Desta forma, existem funções separadas para ambos os modos, emissor e recetor.

Emissor

Na parte do emissor, para garantir a existência de coesão nos dados recebidos são feitas algumas verificações.

Assim, para o envio da informação, o emissor prepara as tramas de início(START) e de fim (END) invocando a função *create_STARTEND_packet().* O envio destas tramas permite ao recetor saber qual o nome do ficheiro a criar e, aquando da receção da trama END, verificar se o tamanho de dados recebidos está correto.

É também no protocolo de aplicação que é adicionado o cabeçalho do pacote (*packet header*) no qual vai a informação relativa ao número do pacote e tamanho de dados que o pacote transporta. Este cabeçalho é útil pois, no final, caso o ficheiro esteja corrompido é possível verificar qual(is) os pacotes que falharam, através do ficheiro de log criado.

Recetor

No recetor apenas é chamada a função *get_message()* que, após executar a função LLREAD, analisa os dados obtidos.

No caso dos Pacotes de Informação (tramas I) é invocada a função *get_only_data().* Esta função remove o cabeçalho e, considerando os valores de L1 e L2, lê os dados existentes na trama.

Já no caso de se tratar de uma trama END, é invocada a função *verify_end()*, que verifica se o tamanho do ficheiro recebido corresponde ao tamanho recebido na trama final, de modo a saber se o ficheiro foi recebido na totalidade.

Validação

Nesta secção são abordados os diversos testes realizados. Foram utilizados no total três tipos de teste, com o propósito de testar a eficiência, para o mesmo ficheiro "pinguim.gif" usado na apresentação do trabalho com um tamanho total de 10968 kilobytes.

Os primeiros tipos de testes realizados tiveram em conta o aumento da percentagem de erros do BCC1 e do BCC2. Os resultados destes testes podem ser verificados anexo.

O segundo tipo de testes efetuados teve em conta diferentes tamanhos dos pacotes enviados. Os resultados destes testes podem ser verificados no anexo.

E por fim, o terceiro tipo de testes realizados foram aplicados a diferentes valores de BAUDRATE. Os resultados destes testes podem ser verificados no anexo.

Para além dos testes apresentados, para testar a lógica do programa e a capacidade de a aplicação conseguir transmitir o ficheiro mesmo que haja anormalidades na porta de série, foram realizados três testes.

O primeiro teste verificava se o mecanismo do timeout funcionava, ou seja, caso o emissor estivesse a mandar a informação na porta de série e esta fosse intencionalmente fechada, o emissor deveria mandar mensagem de timeout em cada X segundos e após Y tentativas o programa fechava com mensagem de erro.

O segundo teste testava a capacidade de a aplicação continuar a correr após a porta de série ser fechada, enquanto o emissor mandava os dados a porta de serie era fechada e antes de dar erro de timeout, a porta voltava a ser aberta e a aplicação continua a mandar a o ficheiro sem qualquer problema.

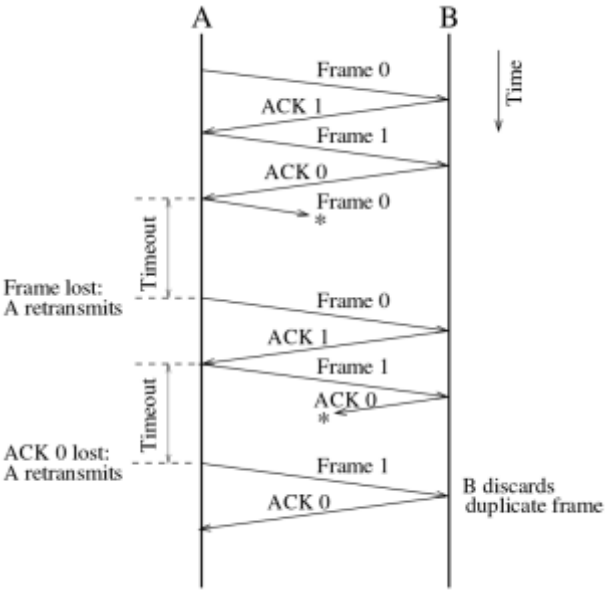
O último teste tinha a ideia de aplicar curtos circuitos na porta de série de forma a que fosse mandado nela valores aleatórios, assim sempre que era dado curto circuito o recetor manda ao emissor mensagem de rejeição REJ até parar de receber valores não esperados.

De forma a verificar também se os pacotes foram todos devidamente enviados, sempre que o programa corre e criado um ficheiro de log que cita todas as respostas do recetor ao emissor e vice-versa, assim e possível verificar se sempre que o recetor manda resposta RR1 o emissor envia uma trama de índice 1.

Eficiência do Protocolo de Ligação de Dados

No mecanismo *Stop and Wait* de ARQ (Automatic Repeat Request), o emissor após enviar uma trama I espera sempre por uma resposta do recetor, se o recetor confirmar a trama enviada o emissor procede e envia uma nova trama, caso contrário o emissor é obrigado a reenviar a trama I até que esta seja aceite pelo recetor. O recetor ao receber uma trama I verifica se deteta algum erro, caso haja erros envia ao emissor mensagem de rejeição NACK, caso contrário manda confirmação ACK ao emissor.

De forma o programa reconhecer se a trama recebida é uma nova ou apenas uma retransmissão, as tramas I e ACK devem ser numeradas do estilo I(0), I(1) e ACK(0), ACK(1), em que ACK(i) indica ao emissor que o recetor está à espera de uma trama I(i).



Este mecanismo necessita do uso de um alarme com timeout de forma a resolver problemas como a perda das tramas I, ACK ou NACK.

O projeto realizado usa este mecanismo de ARQ, sempre que o emissor manda uma trama espera sempre uma resposta com utilização de um alarme de três segundos. Por exemplo, se o emissor enviar uma trama de informação (quer seja Start, End ou normal) espera sempre por um RR (com numeração 0 ou 1 dependendo da numeração da trama enviada) de resposta positiva ou REJ de resposta negativa, isto não se aplica somente as tramas de informação (tramas de supervisão de certa forma usam também este mecanismo) embora apenas estas necessitam necessariamente de serem numeradas.

Tendo em conta os testes efetuados, após os cálculos do débito recebido bits/s e da eficiência de cada teste podemos concluir o impacto que as variáveis de teste têm na eficiência da aplicação.

No gráfico 1 (em anexo) podemos verificar que a percentagem de erros do BCC1 e do BCC2 têm um impacto significativo na eficiência da aplicação, por isso vemos um decréscimo de eficiência à medida que as percentagens de erros aumentam. Neste gráfico também podemos comparar e concluir que os erros do BCC1 causam maior impacto na eficiência, como podemos ver na passagem do ponto '0+0' para o ponto '2+0' em comparação com o ponto '0+2'.

No gráfico 2 (em anexo) podemos concluir que quanto maior forem as tramas de informação mandadas maior a eficiência da aplicação pois menos tramas têm de ser mandadas. Existe um ponto em que o aumento de eficiência estagna, pois, as tramas passam a ter capacidade maior que o tamanho do ficheiro a mandar.

No gráfico 3 (em anexo) verificamos que quanto maior for BAUDRATE, embora seja possível enviar um maior numero de bits pela porta series por segundo, a eficiência diminui.

Conclusão

Os objetivos propostos para este trabalho foram atingidos, tendo sido estabelecido o protocolo com sucesso.

Para tal, foi mantida a independência de camadas entre a parte da aplicação e a parte da ligação, permitindo assim que, alterando uma das camadas, a outra nada ou quase nada tenha que alterar. Deste modo, a camada de aplicação trata apenas da criação e interpretação dos dados de informação e respetivo cabeçalho do pacote de dados, enquanto que a camada de ligação apenas se preocupa com a coesão das tramas recebidas e envio das mesmas com os valores nos cabeçalhos necessários.

A realização deste trabalho permitiu a todos os membros do grupo um melhor entendimento sobre a independência de camadas em sistemas e estruturação e funcionamento de um protocolo de comunicação utilizando a porta de série.

Anexo I

application_layer.h:

```
#ifndef APPLICATION_LAYER_H
#define APPLICATION_LAYER_H

#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 <signal.h>
#include "datalink_layer.h"

#define MODEMDEVICE "/dev/ttyS1"
#define _POSIX_SOURCE 1 /* POSIX compliant source */
#define FALSE 0
#define TRUE 1
#define DATA_CONTROL 1


#define MAXSIZELINK 7 //2*1(bcc2) + 5 other element trama datalink

#define START_PACKET_TYPE 1
#define END_PACKET_TYPE 0


typedef struct {
  int filesize;
  char* filename;
  FILE* fp;
  int size_to_read;
}file_info;

typedef struct{
  int file_descriptor;
  char* status;
}application_layer;


/**
 * @brief Create package to send and send it to the datalink layer using LLWRITE
 *
 * If is not a strat/end frame the package header
 * is added and after that the message is sent
```

```
 * To be used by the Writer
 *
 * @param msg array with the data that is to send
 * @param length length of the message that is to send
 * @return returns TRUE if message was sent and false otherwise
 */
int send_message(unsigned char* msg, int length);

/**
 * @brief Removes the package header from the received trama
 * To be used by the Reader
 *
 *
 * @param readed_msg array with the trama received
 * @param length Pointer to the length of the trama received. This param is updated to the length of the
returned array
 * @return returns an array with the real data from the readed_msg
 */
unsigned char* get_only_data(unsigned char* readed_msg, int* length);

/**
 * @brief Create package to send and send it to the datalink layer using LLWRITE
 *
 * If is not a start/end frame the package header is added and after that the message is sent
 * To be used by the Reader
 *
 * @param msg array with the data that is to send
 * @param length length of the message that is to send
 * @return returns TRUE if message was sent and false otherwise
 */
unsigned char* get_message();

/**
 * @brief Create package to send and send it to the datalink layer using LLWRITE
 *
 * If is not a start/end frame the package header is added and after that the message is sent
 * To be used by the Writer
 *
 * @param msg array with the data that is to send
 * @param length length of the message that is to send
 * @return returns TRUE if message was sent and false otherwise
 */


unsigned char* data_package_constructor(unsigned char* msg, int* length);
/**
 * @brief Get the size of the file set in the global variable file
 * To be used by the both, Reader and Writer
 *
 * @return returns the size of the file
```

```
 */
int get_file_size();
/**
 * @brief Creates a start/end packet array with the name and the size set in the global variable file
 *  To be used by the Writer
 *
 * @param packet array to be filled
 * @param type Type of the packet to be created (START_PACKET_TYPE / END_PACKET_TYPE)
 * @return Returns the size of the packet array created
 */
int create_STARTEND_packet(unsigned char* packet, int type);
/**
 * @brief Read file parameters from the msg and fill the global variable file with that information.
 *  To be used by the Reader
 */
void get_file_params(unsigned char* msg);
/**
 * @brief Read the file by packets of the size specified in the global variable file
 * After that this function calls send_message to send the readed packet
 * To be used by the Writer
 */
void handle_readfile();
/**
 * @brief Write the data to the created file
 * To be used by the Reader
 *
 * @param data Array with the data to be written
 * @param sizetowrite Size of data array
 */
void handle_writefile(unsigned char * data,int sizetowrite);
/**
 * @brief Verify if the end frame has the same parameters from the start frame and from the created file,
mainly the file size
 * @param msg End frame to be checked
 * @return Returns TRUE if is everything ok, and FALSE otherwise
 */
int verify_end(unsigned char* msg);
/**
 * @brief Main function where the program starts
 * Creates the connection calling LLOPEN and starts the execution for reader and for writer
 * @return Returns TRUE if everything is ok and FALSE if can't connect to the serial port
 */
int main(int argc, char** argv);

#endif
```

application_layer.c:

```
#include "application_layer.h"


int is_start = FALSE;
static file_info file;
static application_layer app_info;


int send_message(unsigned char* msg, int length){
   int res;
   if(is_start == FALSE){
      unsigned char* data_package = data_package_constructor(msg, &length);
      res= LLWRITE(app_info.file_descriptor, data_package, &length);

   }
   else{
      is_start= FALSE;
      res = LLWRITE(app_info.file_descriptor, msg, &length);
   }

   if (res == FALSE){

      return FALSE;
   }


   return TRUE;

}


unsigned char* get_message(){
   int length;
   unsigned char* readed_msg;
   unsigned char* only_data;
   static int file_received_size = 0;
      readed_msg = LLREAD(app_info.file_descriptor, &length);
      if(readed_msg == NULL || readed_msg[0] == DISC){
         return readed_msg;
      }
   switch(readed_msg[0]){
      case 0x02:
         fprintf(fp_log, "[BEGIN FILE]\n");
         get_file_params(readed_msg);
         break;
      case 0x01:
         utils_n_package++;
```

```
            only_data = get_only_data(readed_msg, &length);
            handle_writefile(only_data,length);
            file_received_size += length;
            progress_bar(file.filesize, file_received_size, file.filename, 'r');
            break;
        case 0x03:
            fprintf(fp_log, "[END FILE]\n");
            verify_end(readed_msg);
            break;

    }


    return readed_msg;

}

unsigned char* get_only_data(unsigned char* readed_msg, int* length){
    int j=0;
    unsigned int size = readed_msg[2]*256 + readed_msg[3];
    unsigned char* only_data = (unsigned char*) malloc(size);
    for(; j<size; j++){
            only_data[j] = readed_msg[j+4];
    }
    *length = size;
    free(readed_msg);
    return only_data;
}


int verify_end(unsigned char* msg){
    int i=0;
    unsigned char file_size[4];
    int file_size_size = msg[2];
    int file_size_total;

    for(; i<file_size_size; i++){
        file_size[i] = msg[i+3];
    }
    file_size_total = (file_size[0] <<24) | (file_size[1] << 16) | (file_size[2] << 8) | (file_size[3]);

    if(file_size_total == file.filesize && file_size_total == get_file_size()){
            fprintf(fp_log,"[END FILE] Received file size is correct\n");
            return TRUE;
    }
    else{
            fprintf(fp_log,"[END FILE] Received file is probably corrupted\n");
            return FALSE;
    }
    return FALSE;
```

```
}


void get_file_params(unsigned char* msg){

   int i=0;
   int j=0;
   unsigned char filesize[4];
   int filename_size;
   if(msg[1] == 0x00){
      int filesize_size = msg[2];
      for(; i<filesize_size; i++){
         filesize[i] = msg[i+3];
      }
      file.filesize = (filesize[0] <<24) | (filesize[1] << 16) | (filesize[2] << 8) | (filesize[3]);
   }
   i += 3;
   if(msg[i] == 0x01){
      i++;
      filename_size = msg[i];
      i++;
      file.filename = (char*) malloc (filename_size+1);
      for(; j<filename_size; j++,i++){
         file.filename[j] = msg[i];
      }
      file.filename[filename_size] = '\0';
   }

   file.fp = fopen((char*)file.filename,"wb");
   start_counting_time();

}


unsigned char* data_package_constructor(unsigned char* msg, int* length){

      unsigned char* data_package = (unsigned char*) malloc(*length+4);

      unsigned char c = 0x01;
      static unsigned int n = 0;
      int l2 = *length/256;
      int l1 = *length%256;

      data_package[0] = c;
      data_package[1] = (char) n;
      data_package[2] = l2;
      data_package[3] = l1;

      utils_n_package++;
      n++;
```

```
    n = (n % 256);
    int i=0;
    for(; i<*length; i++){
      data_package[i+4] = msg[i];
    }

    *length = *length+4;
    /*free(msg);*/

    return data_package;
}


int main(int argc, char** argv){


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

  if(strcmp("r",argv[2]) != 0 && strcmp("w",argv[2]) != 0){
    printf("Usage:\tinvalid read/write mode. a correct mode (r / w).\n\tex: nserial /dev/ttyS1 r\n");
   exit(1);
  }
  srand(time(NULL));

  app_info.status = argv[2];



  if(strcmp("w", app_info.status)==0){
    if(argv[3] == NULL || argv[4] == NULL){
      printf("You need to specify the file to send and the size to read\n");
      exit(1);
    }
    if(argv[5] == NULL || argv[6] == NULL){
      printf("You need to specify the timeout value and the maximum number of transmissions\n");
      exit(1);
    }
    app_info.file_descriptor = LLOPEN(argv[1], app_info.status, argv[5], argv[6]);
      if(app_info.file_descriptor>0){
        file.filename = (char*) argv[3];
        file.size_to_read = atoi(argv[4]);
        int start_end_max_size;
        unsigned char* start_packet;
        unsigned char* end_packet;
```

```
            file.fp = fopen((char*)file.filename,"rb");
            if(file.fp == NULL)
            {
               printf("invalid file!\n");
               exit(-1);
            }
            if((file.filesize = get_file_size()) == -1)
            {
               return FALSE;
            }
            start_end_max_size = 2*(strlen(file.filename) + 9 ) + MAXSIZELINK; //max size for start/end
package;
            start_packet = (unsigned char*) malloc(start_end_max_size);

            int start_created_size = create_STARTEND_packet(start_packet, START_PACKET_TYPE);

            if(start_created_size == -1){
               printf("Error creating start packet\n");
               exit(-1);
            }

            int i=0;
            for(;i < start_created_size;i++){
            }

            is_start = TRUE;
            if (send_message(start_packet,start_created_size) == FALSE){
               LLCLOSE(app_info.file_descriptor, -1);
            }

            handle_readfile();

            is_start = TRUE;
            end_packet = (unsigned char*) malloc(start_end_max_size);
            int endpacket_size = create_STARTEND_packet(end_packet,END_PACKET_TYPE);

            is_start=TRUE;
            if(send_message(end_packet,endpacket_size) == FALSE){
               LLCLOSE(app_info.file_descriptor, -1);
            }

            LLCLOSE(app_info.file_descriptor,WRITER);


         }
      }
      else if(strcmp("r", app_info.status)==0){
         if(argv[3] == NULL || argv[4] == NULL){
            printf("You need to specify the timeout value and the maximum number of transmissions\n");
            exit(1);
```

```
        }
        app_info.file_descriptor = LLOPEN(argv[1], app_info.status, argv[3], argv[4]);
        if(app_info.file_descriptor>0){
          unsigned char* msg;
          unsigned char null_val[] = {0xAA};


          do{
            msg = get_message();

            if(msg == NULL)
            {
              msg = null_val;
            }

          }while(msg[0] != DISC);

        }
      }else
      {
        printf("Error opening serial port\n");
        return 1;
      }
  return 0;

}


int get_file_size(){

  fseek(file.fp, 0L, SEEK_END);
  int filesize = (int) ftell(file.fp);
  if(filesize == -1)
    return -1;
  fseek(file.fp, 0L, SEEK_SET);
  return filesize;

}

int create_STARTEND_packet(unsigned char* packet, int type){
  int i = 0;
  int j = 3;
  unsigned char filesize_char[4];
  unsigned int filename_length = (unsigned int) strlen(file.filename);

  //convert filesize to and unsigned char array
  filesize_char[0] = (file.filesize >> 24) & 0xFF;
  filesize_char[1] = (file.filesize >> 16) & 0xFF;
  filesize_char[2] = (file.filesize >> 8) & 0xFF;
  filesize_char[3] = file.filesize & 0xFF;
```

```
  //size of filesize unsigned char array, normally is 4
  int length_filesize = sizeof(filesize_char)/sizeof(filesize_char[0]);


  if(type == START_PACKET_TYPE)
    packet[0] = 0x02;
  else if(type == END_PACKET_TYPE) {
    packet[0] = 0x03;
  }
  else{
    return -1;
  }
  packet[1] = 0x00;
  packet[2] = length_filesize;


  //put filesize unsigned char array in packet array
  for(; i < length_filesize; i++,j++){
    packet[j] = filesize_char[i];
  }

  packet[j] = 0x01;
  j++;
  packet[j] =  filename_length;

  j++;
  i=0;
  for(;i < filename_length; i++,j++)
  {
    packet[j] = file.filename[i];
  }


  return j;

}



void handle_readfile()
{
  unsigned char* data = malloc(file.size_to_read);
  int file_sent_size = 0;
  start_counting_time();
  utils_n_package = 0;
  fprintf(fp_log, "[BEGIN FILE]\n");
  fseek(file.fp,0,SEEK_SET);
  while(TRUE)
  {
```

```
      int res = 0;
      res = fread(data,sizeof(unsigned char),file.size_to_read,file.fp);
      if(res > 0)
      {

         if(send_message(data,res) == FALSE){
            LLCLOSE(app_info.file_descriptor, -1);
            exit(-1);
         }
         fprintf(fp_log, "[PACKAGE %d] RR%d\n", utils_n_package, dl_layer.control_value);
         file_sent_size += res;
         progress_bar(file.filesize, file_sent_size, file.filename, 'w');
      }
      if(feof(file.fp))
         break;

   }
   fprintf(fp_log, "[END FILE]\n");

}

void handle_writefile(unsigned char* data,int sizetowrite){

   fseek(file.fp,0,SEEK_END);
   fwrite(data,sizeof(unsigned char),sizetowrite,file.fp);
}
```

datalink_layer.h:

```
#ifndef DATALINK_LAYER_H
#define DATALINK_LAYER_H

#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 <signal.h>
#include <time.h>
#include "utils.h"



#define BAUDRATE B38400
#define MODEMDEVICE "/dev/ttyS1"
```

```c
#define _POSIX_SOURCE 1 /* POSIX compliant source */
#define FALSE 0
#define TRUE 1
#define FLAG 0x7E
#define ADDR 0x03
#define CW 0x03
#define CR 0x07
#define BCCW 0x00
#define BCCR 0x04
#define ERROR_PERCENTAGE_BCC1 0
#define ERROR_PERCENTAGE_BCC2 0
#define S0 0
#define S1 1
#define S2 2
#define S3 3
#define S4 4
#define SESC 5

#define READER 0
#define WRITER 1

#define RR 1
#define REJ 0
#define DISC 0x0B

#define C_START 2
#define C_END 3
#define C_I 0x01


#define TRAMA_S 0
#define TRAMA_I 1

#define ERROR_BCC1 1
#define ERROR_BCC2 2

typedef struct{
   struct termios oldtio;
   struct termios newtio;
   unsigned char control_value;
   unsigned int timeout;
   unsigned int max_transmissions;
} link_layer;

link_layer dl_layer;

/**
 * @brief Handles the alarm from thefunctions that wait a response.
 */
void alarm_handler();
```

```
/**
 * @brief verify the char received and if is ok, put it in the frame(trama)
 * Also verify if BCC1 is incorrect
 *
 * @param c char to check
 * @param state Actual state from the state machine
 * @param trama Frame array to be filled with the char
 * @param actual length of the frame array
 * @param trama_type Type of the trama that is to be check. TRAMA_S or TRAMA_I
 */
void state_machine(unsigned char c, int* state, unsigned char* trama, int* length, int trama_type);.
/**
 * @brief try to establish connection by the writer
 *
 * send the SET command and wait for the UA response
 *
 * @param fd pointer for the serial port descriptor
 * @return returns TRUE if connection is successfully established, and FALSE otherwise
 */
int set_writer(int* fd);
/**
 * @brief try to establish connection by the reader
 *
 * wait a SET command and sned UA as response
 *
 * @param fd pointer for the serial port descriptor
 * @return returns TRUE if connection is successfully established, and FALSE otherwise
 */
int set_reader(int* fd);
/**
 * @brief set and open the serial port with the new settings (ex: BAUDRATE)
 *
 * @param port serial port name
 * @param fd the descriptor that represents the serial port after it has been opened
 */
void set_serial_port(char* port, int* fd);
/**
 * @brief reset serial port settings to the settings from the beggining of the program and close it
 * @param desciptor to the serial port that is to close
 * @returns TRUE if everythings went well
 */
int close_serial_port(int fd);
/**
 * @brief Opens the serial port, and given the role (receiver or sender) prepares to either send a SET and
receive an UA (sender) or
 * receive a SET and send an UA(receiver)
 *
 * @param port name of the port to be opened
 * @param mode the role, either sender ('w') or receiver ('r')
 * @param timeout seconds the alarm activates
```

```
 * @param max_transmissions maximum number of attempts
 * @return int the file descriptor of the port, or -1 in case of error
 */
int LLOPEN(char* port, char* mode, char* timeout, char* max_transmissions);
/**
 * @brief Receives the data layer D1..DN adds BCC2, stuffs it, adds the link layer and creates the complete
packet to be sent
 *
 * @param msg Data of the file
 * @param length lenght of msg
 * @return unsigned char*  the complete packet stuffed to be sent to the receiver
 */
unsigned char* create_frame(unsigned char* msg, int* length);
/**
 * @brief destuffs the control_message and checks if the BCC2 of the control message is equal to the
continuous xor operation of data D1^D2^D3^...DN
 *
 * @param control_message the part stuffed part of a packet
 * @param length the size of control_message
 * @return unsigned char* if the BCC2 is as expected it returns the control_message values after destuffin,
otherwise returns null
 */
unsigned char* verify_bcc2(unsigned char* control_message, int* length);
/**
 * @brief remove frame header and tail
 * @param msg frame array to be changed
 * @param length pointer to the length of the frame array, to be changed at the end
 * @return returns the new frame without header or tail
 */
unsigned char* remove_head_msg_connection(unsigned char* msg, int* length);
/**
 * @brief Adds the link layer to the packet
 *
 * @param stuffed_frame application layer of the packet
 * @param length  size of the application layer of the packet
 * @return unsigned char* the complete packet ready to be sent to the receiver
 */
unsigned char* add_frame_header(unsigned char* stuffed_frame, int* length);
/**
 * @brief does the mechanism of stuffing from the values D1 to Dn and BCC2 of an information packet
 * everytime that the portion of the packet has a flag value 0x7E it is substituted by 0x7D 0x5e, if it finds
0x7D
 * is is substituded by 0x7D 0x5D.
 *
 * @param msg the part of the packet where the stuffing is going to be applied
 * @param length size of msg
 * @return unsigned char*  returns the portion of the packet stuffed
 */
unsigned char* byte_stuffing(unsigned char* msg, int* length);
/**
```

```
 * @brief destuff received frame, basically does the opposite of the function byte_stuffing
 * @param frame to destuff
 * @param length pointer to the length of the frame array, to be changed at the end
 * @return returns the new frame destuffed
 */
unsigned char* byte_destuffing(unsigned char* msg, int* length);
/**
 * @brief writes a new frame to the receiver and waits his response
 *
 * @param fd filed descriptor of the serial port
 * @param msg file data(or other if it is a START or END frame) to be transform into a packet ready to be sent
 * @param length the size of the data
 * @return int TRUE if everything went well otherwise FALSE, for example when timeout occurs more than the maximum times allowed
 */
int LLWRITE(int fd, unsigned char* msg, int* length);
/**
 * @brief Creates and send the response of the receiver, either RR or REJ
 *
 * @param fd descriptor of the serial port
 * @param type describes if the response is to be RR or REJ
 * @param c variable control to know the next type of packet to receive 1 or 0
 * @return int the control value
 */
int send_response(int fd, unsigned int type, unsigned char c);
/**
 * @brief By the ERROR_PERCENTAGE_BCC1 macro this function has a chance to randomly change the value of BCC1 used for testing purposes
 *
 * @param packet the packet to be changed, if its changed at all
 * @param size_packet the size of the packet to change
 * @return unsigned char* the packet either without any changes to it or with a diferrent BCC1
 */
unsigned char* mess_up_bcc1(unsigned char* packet, int size_packet);
/**
 * @brief By ERROR_PERCENTAGE_BCC2 macro this function has a chance to randomly change the value of BCC2, used for testing purposes only
 *
 * @param packet the packet to be changed
 * @param size_packet the size of the packet
 * @return unsigned char* the packet either without any changes or with a different, wrong BCC2
 */
unsigned char* mess_up_bcc2(unsigned char* packet, int size_packet);
/**
 * @brief reads the new frame from the serial port and return it
 * @param fd descriptor for the serial port
 * @ param length pointer to the length of the frame array, to be changed at the end
 * @return returns the readed frame packet
 */
```

```
unsigned char* LLREAD(int fd, int* length);
/**
 * @brief establish the finish set up (sending SET and UA)
 * @param fd descriptor for the serial port
 * @param type type of the mode (w - wirter or r - reader)
 */
void LLCLOSE(int fd, int type);
/**
 * @brief send a disc command and wait a response returning it
 * @param fd descriptor for serial port
 * @return returns the received frame
 */
unsigned char* send_disc(int fd);


#endif
```

datalink_layer.c:

```
/*Non-Canonical Input Processing*/
#include "datalink_layer.h"

volatile int STOP=FALSE;
volatile unsigned char flag_attempts=1;
volatile unsigned char flag_alarm=1;
volatile unsigned char flag_error=0;
volatile unsigned char duplicate=FALSE;
const unsigned char control_values[] = { 0x00, 0x40, 0x05, 0x85, 0x01, 0x81 };


void alarm_handler(){
   fprintf(fp_log, "[ALARM] Timeout\n");
   flag_attempts++;

 if(flag_attempts >= dl_layer.max_transmissions){
   flag_error = 1;
 }
   flag_alarm=1;
}



void state_machine(unsigned char c, int* state, unsigned char* trama, int* length, int trama_type){

   switch(*state){
      case S0:
         if(c == FLAG){
            *state = S1;
```

```
            trama[*length-1] = c;
        }
        break;
    case S1:
        if(c != FLAG){

            trama[*length-1] = c;
            if(*length==4){
                if((trama[1]^trama[2]) != trama[3]){
                    *state = SESC;
                }
                else{
                    *state=S2;
                }
            }
        }
        else
        {
            *length = 1;
            trama[*length-1] = c;
        }
        break;
    case S2:
        trama[*length-1] = c;
        if(c == FLAG){
            STOP = TRUE;
            alarm(0);
            flag_alarm=0;
        }
        else{
            if(trama_type == TRAMA_S){
                *state = S0;
                *length = 0;
            }
        }
        break;
    case SESC:
        trama[*length-1] = c;
        if(c == FLAG){
            if(trama_type == TRAMA_I){
                flag_error = 1;
                STOP = TRUE;
            }
            else{
                *state = S0;
                *length = 0;
            }
        }
  }
}
```

```
int set_writer(int* fd){

 unsigned char SET[5] = {FLAG, ADDR, CW, BCCW, FLAG};
 unsigned char elem;
  int res;
 unsigned char trama[5];
  int trama_length = 0;
 int state=0;
  (void) signal(SIGALRM, alarm_handler);
 while(flag_attempts < dl_layer.max_transmissions && flag_alarm == 1){
      fprintf(fp_log,"[SET CONNECTION] TRY: %x\n", flag_attempts);
      res = write(*fd,SET,5);

    alarm(dl_layer.timeout);
    flag_alarm=0;

   // Wait for UA signal.

    while(STOP == FALSE && flag_alarm == 0){
         res = read(*fd,&elem,1);
         if(res >0) {
            trama_length++;
            state_machine(elem, &state, trama, &trama_length, TRAMA_S);

         }
    }
 }

 if(flag_error == 1){
   fprintf(fp_log,"[SET CONNECTION] Can't connect to the reader\n");
   return FALSE;
 }
 else{
      fprintf(fp_log,"[SET CONNECTION] Successful\n");
  return TRUE;
 }

}

int set_reader(int* fd){

 unsigned char UA[5] = {FLAG, ADDR, CR, BCCR, FLAG};
 char elem;
  int res;
 unsigned char trama[5];
  int trama_length =0;
 int state=0;
 while (STOP==FALSE) {      /* loop for input */
```

```
    res = read(*fd,&elem,1);

    if(res>0){
     trama_length++;
     state_machine(elem, &state, trama, &trama_length, TRAMA_S);

    }
   }

   res = write(*fd,UA,5);

   return TRUE;
}


/* SET Serial Port Initilizations */

void set_serial_port(char* port, int* fd){

   /*
    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 = open(port, O_RDWR | O_NOCTTY );

   if (*fd <0) {perror(port); exit(-1); }

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

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

   /* set input mode (non-canonical, no echo,...) */
   dl_layer.newtio.c_lflag = 0;

   dl_layer.newtio.c_cc[VTIME]    = 1;   /* inter-character timer unused */
   dl_layer.newtio.c_cc[VMIN]     = 0;   /* blocking read until 5 chars received */



   tcflush(*fd, TCIOFLUSH);
```

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

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

}


int close_serial_port(int fd){

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

  close(fd);
  return 0;
}


int LLOPEN(char* port, char* mode, char* timeout, char* max_transmissions){

 int fd;
 int result;
 dl_layer.control_value = 0;
 dl_layer.timeout = atoi(timeout);
 dl_layer.max_transmissions= atoi(max_transmissions);
 set_serial_port(port, &fd);
  open_log_file(mode);

 if(strcmp(mode,"r") == 0){
  result = set_reader(&fd);

 }
 else if(strcmp(mode,"w") == 0){
  result = set_writer(&fd);
 }

 if(result  == TRUE){
  return fd;
 }
 else{
     LLCLOSE(fd, -1);
  return -1;
 }

}
```

```
unsigned char* create_frame(unsigned char* msg, int* length){
   int i=0;
   unsigned char bcc2 = 0x00;
   unsigned char* new_message = (unsigned char*) malloc(*length+1);
   for(i=0; i<*length; i++){
      new_message[i] = msg[i];
      bcc2 ^=msg[i];
   }
   new_message[*length] = bcc2;
      *length = *length+1;
   i=0;
   unsigned char* stuffed_message = byte_stuffing(new_message, length);

   unsigned char* control_message = add_frame_header(stuffed_message, length);

   return control_message;
}



unsigned char* verify_bcc2(unsigned char* control_message, int* length){
   unsigned char* destuffed_message = byte_destuffing(control_message, length);
   int i=0;
   unsigned char control_bcc2 = 0x00;
   for(; i<*length-1; i++){
      control_bcc2 ^= destuffed_message[i];
   }
   if(control_bcc2 != destuffed_message[*length-1]){
      *length = -1;
      return NULL;
   }
   *length = *length-1;
   unsigned char* data_message = (unsigned char*) malloc(*length);
   for(i=0; i<*length; i++){
         data_message[i] = destuffed_message[i];
   }
   free(destuffed_message);
   return data_message;
}



unsigned char* remove_head_msg_connection(unsigned char* msg, int* length){

   unsigned char* control_message = (unsigned char*) malloc(*length-5);
   int i=4;
   int j=0;
   for(; i<*length-1; i++, j++){

      control_message[j] = msg[i];
```

```
    }
    *length = *length-5;
    free(msg);
    return control_message;
}


unsigned char* add_frame_header(unsigned char* stuffed_frame, int* length){
    unsigned char* full_frame = (unsigned char*) malloc(*length+5);
    int i=0;
    full_frame[0] = FLAG;
    full_frame[1] = ADDR;
    full_frame[2] = control_values[dl_layer.control_value];
    full_frame[3] = full_frame[1]^full_frame[2];
    for(; i<*length; i++){
        full_frame[i+4] = stuffed_frame[i];
    }
    full_frame[*length+4] = FLAG;
    *length = *length+5;

    free(stuffed_frame);

    return full_frame;
}


unsigned char* byte_stuffing(unsigned char* msg, int* length){
    unsigned char* str;
    int i=0;
    int j=0;
    unsigned int array_length = *length;
    str = (unsigned char *) malloc(array_length);
    for(; i < *length; i++, j++){

        if(j >= array_length){
            array_length = array_length+(array_length/2);
            str = (unsigned char*) realloc(str, array_length);

        }
        if(msg[i] ==  0x7e){
            str[j] = 0x7d;
            str[j+1] = 0x5e;
            j++;
        }
        else if(msg[i] == 0x7d){
            str[j] = 0x7d;
            str[j+1]= 0x5d;
            j++;
        }
        else{
```

```
            str[j] = msg[i];
        }
    }
    *length = j;
    free(msg);

    return str;
}

unsigned char* byte_destuffing(unsigned char* msg, int* length){
    unsigned int array_length = 133;
    unsigned char* str = (unsigned char*) malloc(array_length);
    int i=0;
    int new_length = 0;

    for(; i<*length; i++){
        new_length++;
        if(new_length >= array_length){
            array_length = array_length+ (array_length/2);
            str = (unsigned char *) realloc(str, array_length);
        }
        if(msg[i] == 0x7d){
            if(msg[i+1] == 0x5e){
                str[new_length-1] = 0x7e;
                i++;
            }
            else if(msg[i+1] == 0x5d){
                str[new_length -1] = 0x7d;
                i++;
            }
        }
        else{
            str[new_length-1] = msg[i];
        }

    }
    *length = new_length;
    free(msg);
    return str;
}



int LLWRITE(int fd, unsigned char* msg, int* length){
    unsigned char* full_message= create_frame(msg, length);
    if(*length<0)
        return FALSE;

    unsigned char elem;
    int res;
```

```
  unsigned char trama[5];
  int trama_length = 0;
  int state=S0;

  STOP=FALSE;
  flag_attempts=1;
  flag_alarm=1;
  flag_error=0;

  while(flag_attempts < dl_layer.max_transmissions && flag_alarm == 1){
    res = write(fd, full_message, *length);

    alarm(dl_layer.timeout);
    flag_alarm=0;
    // Wait for response signal.
    while(STOP == FALSE && flag_alarm == 0){
      res = read(fd,&elem,1);
      if(res >0) {
        trama_length++;
        state_machine(elem, &state, trama, &trama_length, TRAMA_S);

      }
    }

    if (STOP == TRUE) {
      if(trama[2] == control_values[dl_layer.control_value+4]){
        fprintf(fp_log, "[PACKAGE %d] REJ%d\n", utils_n_package, dl_layer.control_value);
        flag_alarm=1;
        flag_attempts = 1;
        flag_error=0;
        STOP = FALSE;
        state = S0;
        trama_length = 0;
      }
    }
  }


  if (flag_error == 1){
    fprintf(fp_log,"[LLWRITE] Flag Error\n");
    return FALSE;
  }

  dl_layer.control_value = dl_layer.control_value^1;
  return TRUE;
}


unsigned char* LLREAD(int fd, int* length){
  unsigned char elem;
```

```
int state = S0;
int res;
*length=0;
flag_error = 0;
STOP = FALSE;
unsigned int msg_array_length = 138;
unsigned char* msg= (unsigned char*) malloc(msg_array_length);
unsigned char* finish = (unsigned char*) malloc(1);
finish[0] = DISC;
while (STOP==FALSE) {      /* loop for read info */
   res = read(fd,&elem,1);
   if(res>0){
      *length = *length+1;
      if(*length >= msg_array_length){
         msg_array_length = msg_array_length+(msg_array_length/2);
         msg = realloc(msg, msg_array_length);
      }
      state_machine(elem, &state, msg, length, TRAMA_I);
   }
}
if((msg[4] == C_I) && (flag_error != 1)){
   msg = mess_up_bcc1(msg, *length);
   msg = mess_up_bcc2(msg, *length);
}

if(flag_error == 1 || msg[2] == CW || msg[2] == CR){
   return NULL;
}
if(msg[2] == DISC){
   fprintf(fp_log, "[LLREAD] DISC received\n");
   LLCLOSE(fd, READER);
   return finish;
}

duplicate = (control_values[dl_layer.control_value] == msg[2]) ? FALSE: TRUE;
unsigned char char2_temp = msg[2];
unsigned char* msg_no_head = remove_head_msg_connection(msg, length);
unsigned char* msg_no_bcc2= verify_bcc2(msg_no_head, length);


if(*length == -1){
   if(duplicate == TRUE){
      send_response(fd, RR, char2_temp);
      return NULL;
   }
   else{
      send_response(fd, REJ, char2_temp);
      return NULL;
   }
}
```

```
    else{
        if(duplicate != TRUE){
            dl_layer.control_value = send_response(fd, RR, char2_temp);
            return msg_no_bcc2;
        }
        else{
            send_response(fd, RR, char2_temp);
            return NULL;
        }

    }

}

int send_response(int fd, unsigned int type, unsigned char c){
    unsigned char bool_val;
    unsigned char response[5];
    response[0] = FLAG;
    response[1] = ADDR;
    response[4] = FLAG;
    if(c == 0x00)
        bool_val = 0;
    else
        bool_val = 1;

    switch (type) {
        case RR:
            if (utils_n_package >= 1)
                fprintf(fp_log, "[PACKAGE %d] RR%d\n", utils_n_package, bool_val^1);
            utils_response_value[0] = RR;
            utils_response_value[1] = bool_val^1;
            response[2] = control_values[(bool_val^1)+2];
            break;
        case REJ:
            if (utils_n_package >= 1)
                fprintf(fp_log, "[PACKAGE %d] REJ%d\n", utils_n_package, bool_val);
            utils_response_value[0] = REJ;
            utils_response_value[1] = bool_val;
            response[2] = control_values[bool_val+4];
            break;
    }
    response[3] = response[1]^response[2];

    write(fd, response, 5);

    return bool_val^1;
}


void LLCLOSE(int fd, int type){
```

```
    unsigned char * received;
    if(type == READER){
        received = send_disc(fd);
        fprintf(fp_log,"[LLCLOSE] DISC sended with success \n");
        if(received[2] == CR){
            fprintf(fp_log,"[LLCLOSE] Final UA received with success\n");
        }else {
            fprintf(fp_log,"[LLCLOSE] Problem receiving final UA\n");
        }


    }else if(type == WRITER){

        received = send_disc(fd);
        fprintf(fp_log,"[LLCLOSE] DISC sended with success \n");
        if(received[2] == DISC){
            fprintf(fp_log,"[LLCLOSE] Final DISC received with success\n");
        }else {
            fprintf(fp_log,"[LLCLOSE] Problem receiving final DISC\n");
        }
        unsigned char UA[5] = {FLAG, ADDR, CR, BCCR, FLAG};
        write(fd,UA,5);
        fprintf(fp_log,"[LLCLOSE] Final UA sended with success\n");
        sleep(1);
    }

    close_serial_port(fd);
    fprintf(fp_log, "\n\n");
    fclose(fp_log);
}

unsigned char* send_disc(int fd){

    unsigned char disc[5] = {FLAG, ADDR, DISC, ADDR^DISC, FLAG};
    unsigned char elem;
    int res;
    unsigned char* trama = (unsigned char*) malloc(5);
    int trama_length = 0;
    int state=0;
    flag_attempts=1;
    flag_alarm=1;
    flag_error=0;
    STOP = FALSE;

    while((flag_attempts < dl_layer.max_transmissions) && (flag_alarm == 1)){
        res = write(fd,disc,5);

        fprintf(fp_log,"[LLCLOSE] Send DISC: try %d \n", flag_attempts);
        alarm(dl_layer.timeout);
        flag_alarm=0;
```

```
        while(STOP == FALSE && flag_alarm == 0){
        res = read(fd,&elem,1);
              if(res >0) {
                 trama_length++;
                    state_machine(elem, &state, trama, &trama_length, TRAMA_S);
              }
        }

   }

   return trama;
}

unsigned char* mess_up_bcc1(unsigned char* packet, int size_packet){

   unsigned char* messed_up_msg = (unsigned char*) malloc(size_packet);
   unsigned char letter;

   memcpy(messed_up_msg, packet, size_packet);
   int perc = (rand()%100)+1;
   if(perc <= ERROR_PERCENTAGE_BCC1){
      do{
         letter = (unsigned char) ('A' + (rand()%256));
      }while(letter == messed_up_msg[3]);
      messed_up_msg[3] = letter;
      flag_error=1;
      printf("BBC1 messedUP\n");
   }
   free(packet);
   return messed_up_msg;
}


unsigned char* mess_up_bcc2(unsigned char* packet, int size_packet){

   unsigned char* messed_up_msg = (unsigned char*) malloc(size_packet);
   unsigned char letter;

   memcpy(messed_up_msg, packet, size_packet);
   int perc = (rand() %100)+1;

   if(perc<= ERROR_PERCENTAGE_BCC2){
      //change data to have error in bcc2
      int i = (rand() % (size_packet-5))+4;
      do{
         letter = (unsigned char) ('A' + (rand()%256));
      }while(letter == messed_up_msg[i]);
      messed_up_msg[i] = letter;
      printf("Data messedUP\n");
```

```
    }
    free(packet);
    return messed_up_msg;
}
```

Anexo II

Tabelas dos testes efetuados

VARIAR % ERROS (FER)

Nº total de bytes	10968
Nº total de bits	87744
C (Baudrate)	38400
Tamanho de cada pacote (bytes)	128

Probabilidade de erro (%bcc1 +%bcc2)	Tempo (s)	R (bits/tempo)	S (R/C)
0+0	3,4626	25340,49558	0,659908739
2+0	6,5564	13382,95406	0,348514429
0+2	3,5368	24808,86677	0,646064239
2+2	6,5983	13297,97069	0,34630132
4+2	12,8684	6818,563302	0,177566753
2+4	12,7086	6904,301025	0,179799506
4+4	12,9483	6776,488033	0,176471043
6+4	16,0422	5469,573999	0,142436823
4+6	13,0295	6734,256879	0,175371273
6+6	19,1863	4573,263214	0,119095396
8+6	31,5679	2779,532373	0,072383656
6+8	25,3904	3455,794316	0,089994644
8+8	34,7148	2527,567493	0,06582207
10+8	47,3332	1853,751701	0,048274784
8+10	37,8529	2318,025832	0,060365256
10+10	50,1379	1750,053353	0,045574306

Tabela 1- Resultados dos testes de variação da % erros dos BCCs

VARIAR TAMANHO DAS TRAMAS

Nº total de bytes	10968
Nº total de bits	87744
C (Baudrate)	38400

Tamanho de cada pacote(bytes)	tempo (s)	R (bits/tempo)	S (R/C)
8	12,0654	7272,365607	0,189384521
8	12,072	7268,389662	0,189280981
16	7,5097	11684,08858	0,30427314
16	7,4935	11709,3481	0,30493094
32	5,2379	16751,75166	0,436243533
32	5,2177	16816,60502	0,437932422
64	4,0505	21662,5108	0,564127885
64	4,0604	21609,69363	0,562752438
128	3,4614	25349,28064	0,660137517
128	3,4583	25372,00359	0,66072926
256	3,1697	27682,11503	0,720888412
256	3,1692	27686,48239	0,721002146
512	3,0195	29059,11575	0,756747806
512	3,0182	29071,6321	0,757073753
1024	2,9452	29792,20426	0,775838653
1024	2,9455	29789,16992	0,775759633
10968	2,8769	30499,49599	0,794257708
10968	2,8765	30503,73718	0,794368156
131072	2,877	30498,43587	0,794230101
131072	2,877	30498,43587	0,794230101

Tabela 2 - Resultados dos testes de diferentes tamanhos de tramas

VARIAR C (BAUDRATE)

Nº total de bytes	10968
Nº total de bits	87744
Tamanho de cada pacote (bytes)	128

C (Baudrate)	tempo (s)	R (bits/tempo)	S (R/C)
600	205,4266	427,1306637	0,71188444
600	206,147	425,6380156	0,709396693
1200	102,7849	853,6662486	0,711388541
1200	102,7848	853,6670792	0,711389233
1800	68,5738	1279,555749	0,710864305
1800	68,5733	1279,565079	0,710869488
2400	52,9261	1657,858788	0,690774495
2400	52,9272	1657,824332	0,690760138
4800	26,5564	3304,062298	0,688346312
4800	26,5583	3303,825923	0,688297067
9600	13,376	6559,808612	0,683313397
9600	13,3717	6561,918081	0,683533133
19200	6,7767	12947,89499	0,674369531
19200	6,7863	12929,57871	0,673415558
38400	3,4666	25311,25599	0,659147291
38400	3,4661	25314,90724	0,659242376

Tabela 3 - Resultados dos testes com diferentes valores de BAUDRATE

Gráficos dos testes efetuados

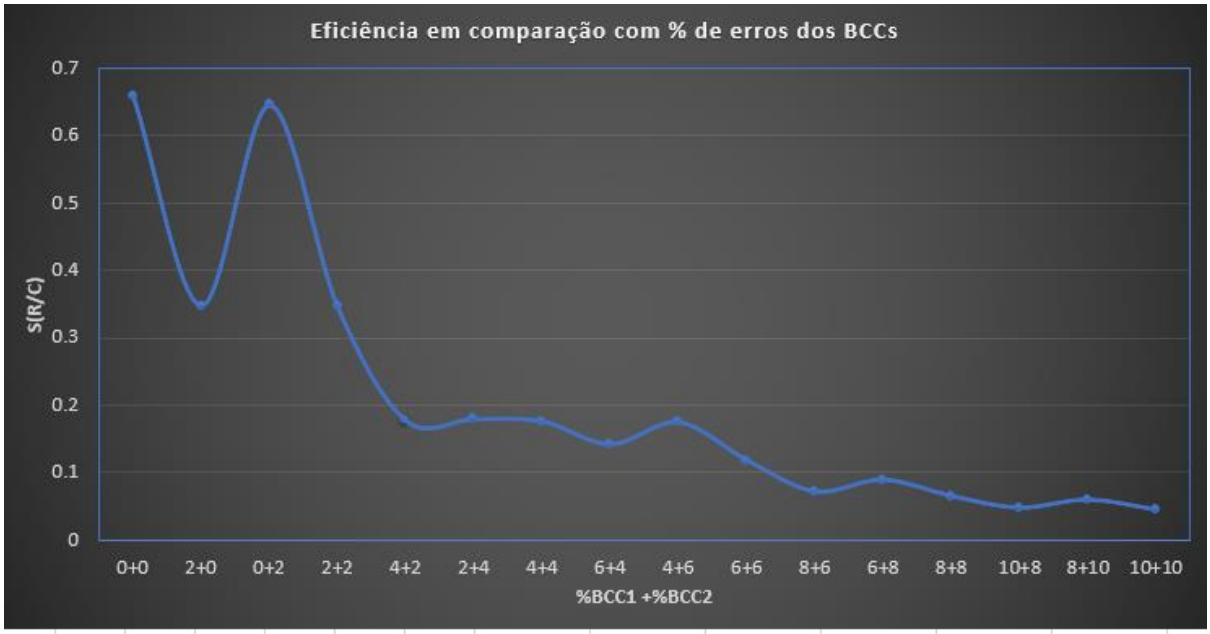


Figura 1- Gráfico eficiência, percentagem de erros obtido dos dados da tabela 1.

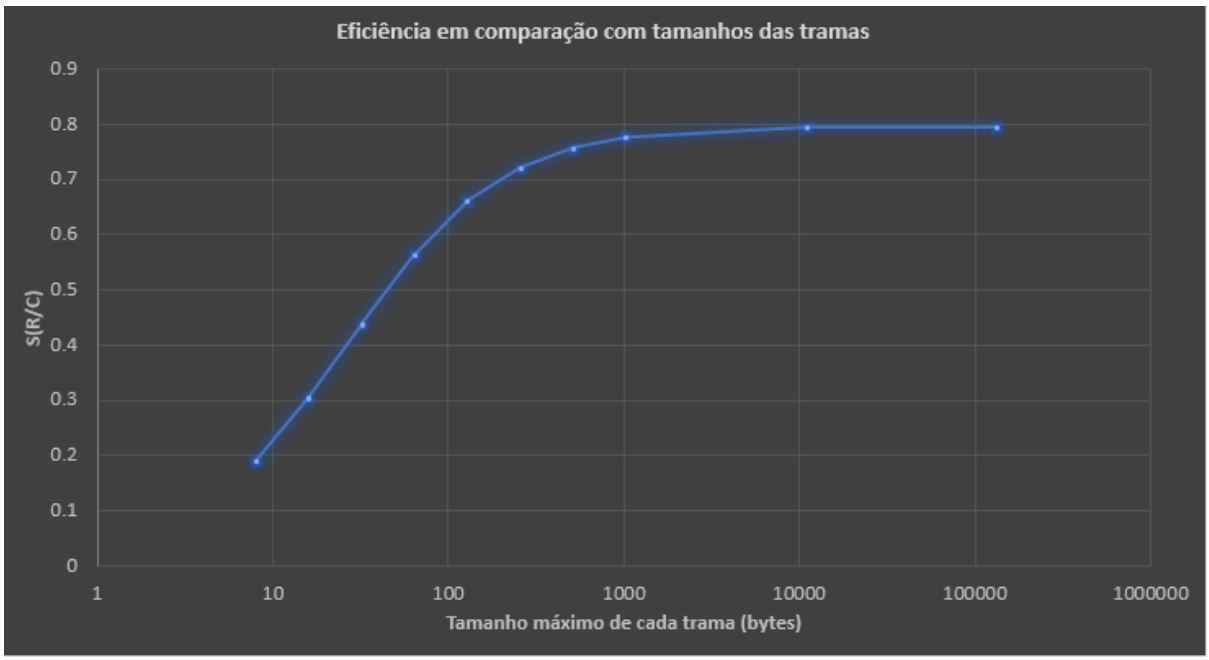


Figura 2- Gráfico de eficiência em relação ao tamanho das tramas obtido do dados da tabela 2.

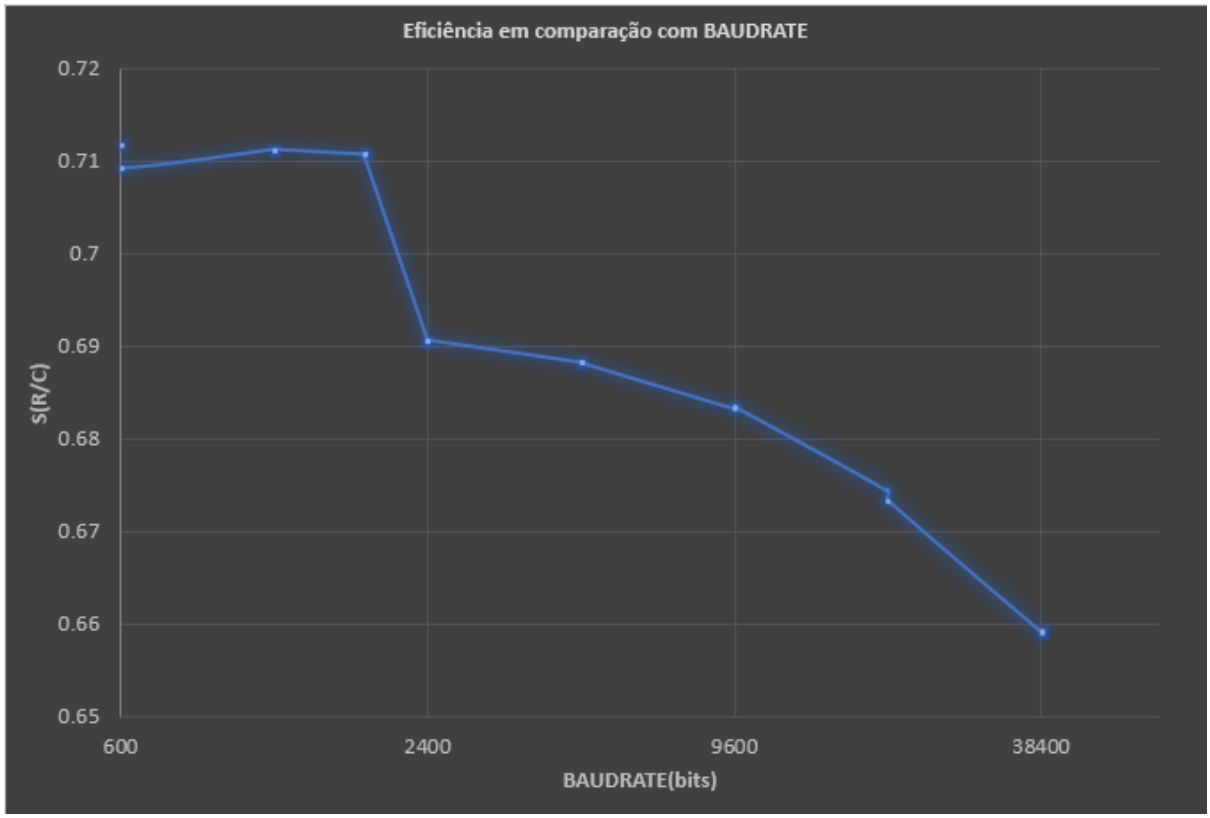


Figura 3 - Gráfico de eficiência em comparação com BAUDRATE retirado dos dados da tabela 3