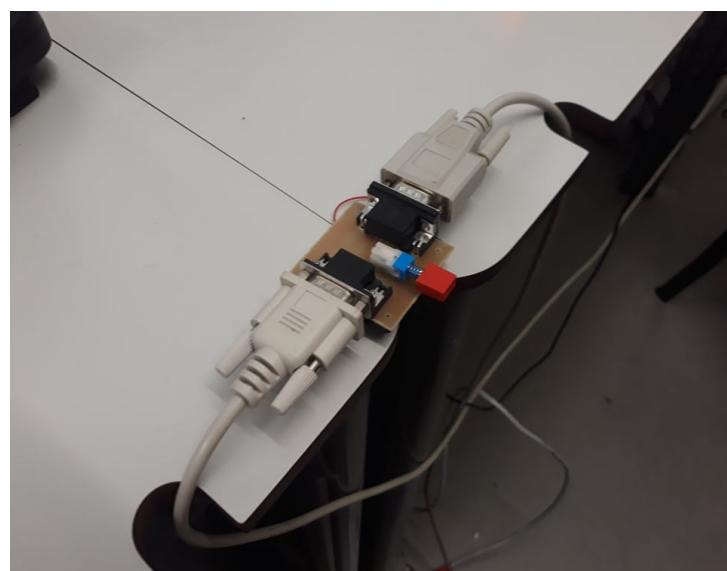


# 1º Trabalho Laboratorial

## Ligaçāo de Dados



**Trabalho realizado por:**

Diogo Machado up201706832@fe.up.pt  
Eduardo Ribeiro up201705421@fe.up.pt  
José Guerra up201706421@fe.up.pt

**Unidade Curricular:** RCOM

**Regente:** Manuel Alberto Pereira Ricardo

**Ano Letivo:** 2019/2020

**Data de Entrega:** 28 de outubro de 2019

# Sumário

Este relatório foi realizado no âmbito da unidade curricular redes de computadores (RCOM) do 3º ano do mestrado integrado em engenharia informática e de computação (MIEIC). O relatório incide sobre o primeiro trabalho laboratorial realizado cujo foco é a transferência de dados através de uma aplicação. A transferência de dados é feita através da implementação de um protocolo de comunicação entre duas máquinas. O trabalho prático foi concluído com sucesso sendo todos os objetivos definidos no início alcançados.

O relatório serve para detalhar a implementação do trabalho bem como explicar os conceitos teóricos que foram aplicados.

## Introdução

O objetivo deste trabalho pode subdividir em dois objetivos concretos, o objetivo relativo a camada do protocolo de ligação de dados e o objetivo relativo a camada da aplicação. O principal objetivo do protocolo de Ligação de Dados é fornecer um serviço de comunicação de dados fiável entre dois sistemas ligados por um meio (canal) de transmissão – neste caso, um cabo série. O objetivo da aplicação é desenvolver um protocolo de aplicação muito simples para transferência de um ficheiro, usando o serviço fiável oferecido pelo protocolo de ligação de dados.

O objetivo do relatório é explicar ao leitor a parte teórica deste trabalho, bem como a nossa implementação do que foi proposto pelo guião, tendo a seguinte estrutura:

- Arquitetura - Visualização dos blocos funcionais e interfaces.
- Estrutura do código - Representação das APIs, principais estruturas de dados, principais funções e sua relação com a arquitetura.
- Casos de uso principais - Identificação dos mesmos bem com as sequências de chamada de funções.
- Protocolo de ligação lógica - Identificação dos principais aspectos funcionais bem como a descrição da estratégia de implementação destes aspectos com apresentação de extratos de código.
- Protocolo de aplicação - Identificação dos principais aspectos funcionais bem como a descrição da estratégia de implementação destes aspectos com apresentação de extratos de código.
- Validação - Descrição dos testes efetuados com apresentação quantificada dos resultados.
- Eficiência do protocolo de ligação de dados - Caracterização estatística da eficiência do protocolo, feita com recurso a medidas sobre o código desenvolvido.
- Conclusões - Síntese da informação apresentada nas secções anteriores e reflexão sobre os objetivos de aprendizagem alcançados.

# Arquitetura

O trabalho está dividido em dois blocos: o writer (emissor) e o reader (leitor). Ambos os blocos chamam funções presentes na camada de aplicação e na camada de ligação de dados porém há uma clara distinção de quem está a utilizar as estas camadas, logo embora ambos utilizem o `Ilopen()` e `Ilclose()` (funções da aplicação de dados) a implementação destas funções garante que haja independência entre emissor e leitor.

## Estrutura de código

O código encontra-se dividido em dois ficheiros principais (mains): o `WritenonCanonical.c` que corresponde ao writer; o `noncanonical.c` que corresponde ao reader. Ambos estes ficheiros chamam funções da aplicação e esta por sua vez chama funções da ligação de dados.

### **writenoncanonical (writer)**

#### Funções principais da camada de ligação

- `Ilopen()` - envia trama de supervisão *SET* e recebe trama *UA*
- `Ilwrite()` - efetua *byte stuffing* das *I-frames* e envia-as para o recetor
- `Ilclose()` - envia trama de supervisão *DISC*, recebe trama *DISC* e envia trama *UA*

#### Funções principais da camada de aplicação

- `sendFile()` - abre o ficheiro, separa-o em pacotes, envia-os dentro de tramas para o recetor e fecha o ficheiro
- `openFile()` - efetua a chamada ao sistema que abre o ficheiro a enviar
- `closeFile()` - fecha o ficheiro após o seu envio
- `getFileSize()` - obtém o tamanho de um ficheiro aberto
- `buildControlPacket()` - constrói um pacote de controlo, para ser enviado numa *I-frame*
- `buildDataPacket()` - constrói um pacote de dados do ficheiro, para ser enviado numa *I-frame*

#### Estruturas de dados

```
struct linkLayer {
    char port[20]; /* Dispositivo /dev/ttySx, x = 0, 1 */
    int baudRate; /* Velocidade de transmissão */
    unsigned int sequenceNumber; /* Número de sequência da trama: 0, 1 */
    unsigned int timeout; /* Valor do temporizador: 1 s */
    unsigned int numTransmissions; /* Número de tentativas em caso de falha */
    unsigned char frame[MAX_SIZE_FRAME]; /* Trama */
    unsigned int frameLength; /* Comprimento atual da trama */
};
```

```

struct applicationLayer {
    int fileDescriptor; /* Descritor correspondente à porta série */
    int status; /* TRANSMITTER | RECEIVER */
};

```

### Variáveis globais

- **struct linkLayer ll;**
- **struct applicationLayer al;**
- **struct termios oldtio;**

### Macros pertinentes:

- **TRANSMITTER**
- **MAX\_PACK\_SIZE**
- **MAX\_DATA\_SIZE**
- **CTRL\_START**
- **CTRL\_DATA**
- **CTRL\_END**

## **noncanonical (reader)**

### Funções principais da camada de ligação

- **llopen()** - recebe trama de supervisão *SET* e envia trama *UA*
- **lread()** - recebe as *I-frames*, lê-as, e efetua *byte destuffing*
- **llclose()** - recebe trama de supervisão *DISC*, envia trama *DISC*, e recebe trama *UA*

### Funções principais da camada de aplicação

- **receiveFile()** - recebe o ficheiro, recebe as tramas e guarda-os pacotes no novo ficheiro
- **openFile()** - efetua a chamada ao sistema que cria o ficheiro onde vai ser guardada a informação recebida
- **closeFile()** - fecha o novo ficheiro após recebê-lo na totalidade
- **getFileSize()** - obtém o tamanho de um ficheiro aberto
- **parseControlPacket()** - extrai a informação de um pacote de controlo de uma *I-frame*
- **parseDataPacket()** - extrai a informação de um pacote de dados do ficheiro de uma *I-frame*

Estruturas de dados (igual a **writenoncanonical (writer)**, acima)

```
struct linkLayer {
    char port[20]; /* Dispositivo /dev/ttySx, x = 0, 1 */
    int baudRate; /* Velocidade de transmissão */
    unsigned int sequenceNumber; /* Número de sequência da trama: 0, 1 */
    unsigned int timeout; /* Valor do temporizador: 1 s */
    unsigned int numTransmissions; /* Número de tentativas em caso de falha */
    unsigned char frame[MAX_SIZE_FRAME]; /* Trama */
    unsigned int frameLength; /* Comprimento atual da trama */
};

struct applicationLayer {
    int fileDescriptor; /* Descritor correspondente à porta série */
    int status; /* TRANSMITTER | RECEIVER */
};
```

### Variáveis globais

- **struct linkLayer ll;**
- **struct applicationLayer al;**
- **struct termios oldtio;**

### Macros pertinentes

- **RECEIVER**
- **MAX\_PACK\_SIZE**
- **MAX DATA SIZE**
- **CTRL\_START**
- **CTRL\_DATA**
- **CTRL\_END**

# Casos de Uso Principais

## Interface

A interface permite ao transmissor escolher o ficheiro a enviar.

O utilizador, utilizando a consola, correrá o programa, dando um conjunto de argumentos. Do lado do emissor, deve ser inserida a porta de série a ser utilizada (ex. **/dev/ttyS0**) e o nome do ficheiro a ser enviado (ex. **pinguim.gif**). Do lado do receptor, apenas deve ser inserida a porta de série.

## Transmissão de Dados

A transmissão de dados ocorre via porta de série, entre dois computadores, dando-se a seguinte sequência de eventos:

- O utilizador, do lado do transmissor, escolhe o ficheiro a ser enviado;
- É configurada a ligação entre os dois computadores;
- A ligação é estabelecida;
- O transmissor envia os dados, trama a trama;
- Simultaneamente, o receptor recebe os dados, trama a trama;
- À medida que recebe os dados, o receptor guarda-os num ficheiro com o mesmo nome do ficheiro enviado pelo emissor;
- A ligação é terminada.

# Protocolos utilizados

## Protocolo de ligação de dados

No protocolo de ligação lógica, tal como é pedido no guião do trabalho, foram implementadas as funções **Ilopen**, **Ilwrite**, **Ilread** e **Ilclose**. São as principais funções deste protocolo, e servem de interface para o protocolo de aplicação usar as funcionalidades do data link. Estas funções, respetivamente, fazem: o estabelecimento da ligação entre o emissor e o recetor; o envio de uma trama de informação, até este ter sucesso; a receção de uma trama de informação, até este ter sucesso; e a finalização da ligação entre o emissor e o recetor. As leituras de qualquer trama são feitas através de uma **máquina de estados**, que vai recebendo byte a byte a mensagem e executando mudanças de estado, de modo a que apenas se chega ao estado final se a trama recebida tiver um formato válido.

### LLOPEN

```
/**  
 * Function that opens and establishes the connection between the receiver and the transmitter  
 * @param port Port name  
 * @param role Flag that indicates the transmitter or the receiver  
 * @return File descriptor; -1 in case of error  
 */  
int llopen(char* port, int role);
```

Esta função começa por preencher os campos da estrutura *LinkLayer* com os valores corretos, abrir a ligação à porta de série e criar o descritor de ficheiro, e por instalar o alarm handler de modo a poder dar os timeouts. De seguida, tendo em conta o valor do parâmetro **role**, chama uma das duas funções auxiliares, **llOpenTransmitter** e **llOpenReceiver**.

```
/**  
 * Opens the connection for the transmitter  
 * @param fd File descriptor for the serial port  
 * @return File descriptor; -1 in case of error  
 */  
int llOpenTransmitter(int fd);
```

```
/**  
 * Opens the connection for the receiver  
 * @param fd File descriptor for the serial port  
 * @return File descriptor; -1 in case of error  
 */  
int llOpenReceiver(int fd);
```

Estas funções desempenham o resto das funcionalidades que o emissor e recetor deveriam ter no **Ilopen**. O emissor envia um trama de supervisão SET e fica a espera de ler um trama UA, ocorrendo o timeout e/ou a saída do programa se o número máximo de retransmissões for ultrapassado. O recetor lê um trama SET, e depois de o receber envia um trama UA. O envio de tramas é feito com a função **sendFrame**, e receção com **readSupervisionFrame**. As tramas são geradas através da função **createSupervisionFrame**. A receção das tramas é feita byte a byte, e o valor de cada um é processado através de uma máquina de estados, através da função **changeState**. O envio é feito trama a trama.

## LLWRITE

```
/**  
 * Function that writes the information contained in the buffer to the serial port  
 * @param fd File descriptor of the serial port  
 * @param buffer Information to be written  
 * @param length Length of the buffer  
 * @return Number of characters written; -1 in case of error  
 */  
int llwrite(int fd, unsigned char* buffer, int length);
```

Esta função é apenas chamada pelo emissor, de modo a enviar tramas de informação ao receptor, e é composta por várias fases:

- Criação da trama de informação, utilizando os dados da aplicação passados como argumento (**createInformationFrame**);
- Introdução de *byte stuffing* na trama gerada (**byteStuffing**);
- Envio da trama, com o número de sequência correto (S\_0 ou S\_1) (**sendFrame**);
- Leitura da resposta (RR ou REJ), com possibilidade de timeout (**readSupervisionFrame**);
- Saída da função, se foi recebido um RR, ou reenvio da trama, se for recebido um REJ, as vezes que forem necessárias até receber uma confirmação positiva.

O timeout das tramas I, SET e DISC é feito da seguinte maneira. **finish** e **resendFrame** são flags modificadas pelo alarm handler, quando ocorre um timeout, que indicam se o programa deve acabar ou se a trama deve ser reenviada, respetivamente.

```
while (finish != 1)  
{  
    read_value = readSupervisionFrame(responseBuffer, fd, wantedBytes, 2, END_SEND);  
  
    if (resendFrame)  
    {  
        sendFrame(ll.frame, fd, ll.frameLength);  
        resendFrame = false;  
    }  
  
    if (read_value >= 0)  
    { // read_value é o índice do wantedByte que foi encontrado  
        // Cancels alarm  
        alarm(0);  
        finish = 1;  
    }  
}  
  
if (read_value == -1)  
{  
    printf("Closing file descriptor\n");  
  
    closeNonCanonical(fd, &oldtio);  
    return -1;  
}
```

## LLREAD

```
/**  
 * Function that reads the information written in the serial port  
 * @param fd File descriptor of the serial port  
 * @param buffer Array of characters where the read information will be stored  
 * @return Number of characters read; -1 in case of error  
 */  
int llread(int fd, unsigned char* buffer);
```

Esta função é apenas chamada pelo recetor, de modo a receber tramas de informação provenientes do emissor, e é composta por várias fases:

- Leitura da trama de informação enviada pelo emissor (**readInformationFrame**);
- Utilização de *byte destuffing* na trama recebida (**byteDestuffing**);
- Verificação do BCC2 (**createBCC2**, e comparação com o BCC2 recebido na trama);
- Determinação da resposta que deve ser dada ao emissor, tendo em conta o resultado dos passos anteriores;
- Criação e envio da trama de supervisão correta ao emissor (RR ou REJ, com o número de sequência de trama correto) (**createSupervisionFrame** e **sendFrame**);
- Saída da função, se tudo funcionou corretamente e o buffer passado como argumento foi preenchido, ou tentativa de nova leitura de uma trama de informação, se algum erro ocorreu.

## LLCLOSE

```
/**  
 * Function that closes the connection between the receiver and the transmitter  
 * @param fd File descriptor of the port  
 * @param role Flag that indicates the transmitter or the receiver  
 * @return Positive value when sucess; negative value when error  
 */  
int llclose(int fd, int role);
```

Tal como o llopen, esta função recebe um parâmetro **role**, que indica se é o emissor ou recetor a chamar a função.

```
/**  
 * Closes the connection for the receiver  
 * @param fd File descriptor for the serial port  
 * @return Positive value when sucess; negative value when error  
 */  
int llCloseReceiver(int fd);
```

```
/**  
 * Closes the connection for the transmitter  
 * @param fd File descriptor for the serial port  
 * @return Positive value when sucess; negative value when error  
 */  
int llCloseTransmitter(int fd);
```

O emissor envia um DISC, recebe um DISC do recetor e envia um UA, fechando a ligação através da porta de série. O recetor recebe o DISC do emissor, envia um DISC, e recebe um UA, fechando também a ligação posteriormente se não houver erros. Foi implementada a possibilidade de ocorrência de timeouts nas transmissões corretas.

## FUNÇÕES AUXILIARES

Para além das funções descritas acima, várias outras funções auxiliares foram implementadas, que podem ser observadas melhor nos anexos. Como já foi dito, as funções **readSupervisionFrame** e **readInformationFrame** fazem a leitura da trama utilizando uma máquina de estados, que pode ser melhor observada nos anexos, nos ficheiros statemachine.c e statemachine.h.

## Protocolo de aplicação

O protocolo da aplicação tem como funções principais as funções **sendFile** e **receiveFile**, que são chamadas nas funções main do emissor e do recetor, respetivamente. Estas funções fazem todas as tarefas que cada máquina deve executar.

```
/**  
 * Function to send a file, using the serial port, to its destination  
 * @param port Name of the serial port  
 * @param fileName Name of the file to be sent  
 * @return 0 if successful; negative if an error occurs  
 */  
int.sendFile(char *port , char* fileName);
```

```
/**  
 * Function to receive a file, sent through the serial port  
 * @param port Name of the serial port  
 * @return 0 if successful; negative if an error occurs  
 */  
int.receiveFile(char *port);
```

## SENDFILE

São a seguir listadas as principais funcionalidades da função **sendFile**:

- Abertura do ficheiro a ser enviado;
- Estabelecimento da ligação com o recetor, utilizando a função **Ilopen**;
- Envio do pacote de controlo START, usando **Ilwrite** (contém o nome e tamanho do ficheiro a enviar);
- Fragmentação do ficheiro em pequenos pacotes, cujo tamanho é dependente do tamanho máximo especificado para os pacotes de dados, e envio de cada pacote, por ordem, ao recetor (monitorizando a ordem como o número de sequência), utilizando **Ilwrite**;
- Envio do pacote de controlo END, usando **Ilwrite**;
- Finalização da ligação com o recetor, usando **Ifclose**.

## RECEIVEFILE

São a seguir listadas as principais funcionalidades da função receiveFile:

- Estabelecimento da ligação com o emissor, utilizando a função **llopen**;
- Receção do pacote de controlo START, usando **llread** (contém o nome e tamanho do ficheiro a enviar);
- Abertura de um novo ficheiro com o nome idêntico ao que foi passado no pacote de controlo START;
- Receção, pacote a pacote, com **llread**, dos fragmentos de ficheiro enviados sequencialmente pelo emissor (sendo que a ordem é confirmada através do número de sequência). Cada fragmento é escrito no ficheiro aberto, de modo a juntar os fragmentos recebidos para recriar o ficheiro;
- Fragmentação do ficheiro em pequenos pacotes, cujo tamanho é dependente do tamanho máximo especificado para os pacotes de dados, e envio de cada pacote, por ordem, ao receptor (monitorizando a ordem como o número de sequência), utilizando **llwrite**;
- Receção do pacote de controlo END, usando **llread**;
- Comparação do tamanho do ficheiro recriado com o tamanho de ficheiro passado nos pacotes de controlo;
- Comparação das informações passadas nos pacotes de controlo START e END, para detetar quaisquer possíveis erros no envio por parte do emissor;
- Finalização da ligação com o emissor, usando **llclose**.

## FUNÇÕES AUXILIARES

Para além das funções descritas acima, foram feitas algumas funções auxiliares, mais precisamente para a construção dos pacotes de controlo e de dados (**buildDataPacket** e **buildControlPacket**), e para o parse destes pacotes, de modo a retirar as informações dos mesmos (**parseControlPacket** e **parseDataPacket**). Foram também feitas funções auxiliares de manipulação de ficheiros (**getFileSize**, **openFile** e **closeFile**).

## Informações Gerais

Para todas as funções descritas, quer na camada de ligação de dados, quer na camada da aplicação, é verificado o valor de retorno, ocorrendo o tratamento de possíveis erros.

# Validação

## Testes efetuados

De forma a verificar o comportamento do programa, os seguintes testes foram efetuados:

- Envio de vários ficheiros, com diferentes tamanhos
- Interrupção da ligação por cabo entre as portas de série
- Geração de ruído na ligação entre as portas de série
- Envio de ficheiros com diferentes taxas de simulação de erros
- Envio de ficheiros com diferentes valores de *baudrate* (capacidade de ligação)
- Envio de ficheiros com diferentes tamanhos para as *I-frames*
- Envio de ficheiros com a simulação de diferentes tempos de propagação de *I-frames*

## Resultados obtidos

Todos os testes que foram efetuados foram concluídos com sucesso, verificando-se o comportamento que era esperado.

## Eficiência do protocolo de ligação de dados

De modo a avaliar a eficiência do programa, foram efetuados os testes em seguida explicitados.

Cada valor obtido vem da média de 6 ensaios para as condições em questão, sendo eliminados os dois valores mais extremos (o menor e o maior), obtendo-se assim uma média de 4 valores, para minimizar o efeito de anomalias estatísticas.

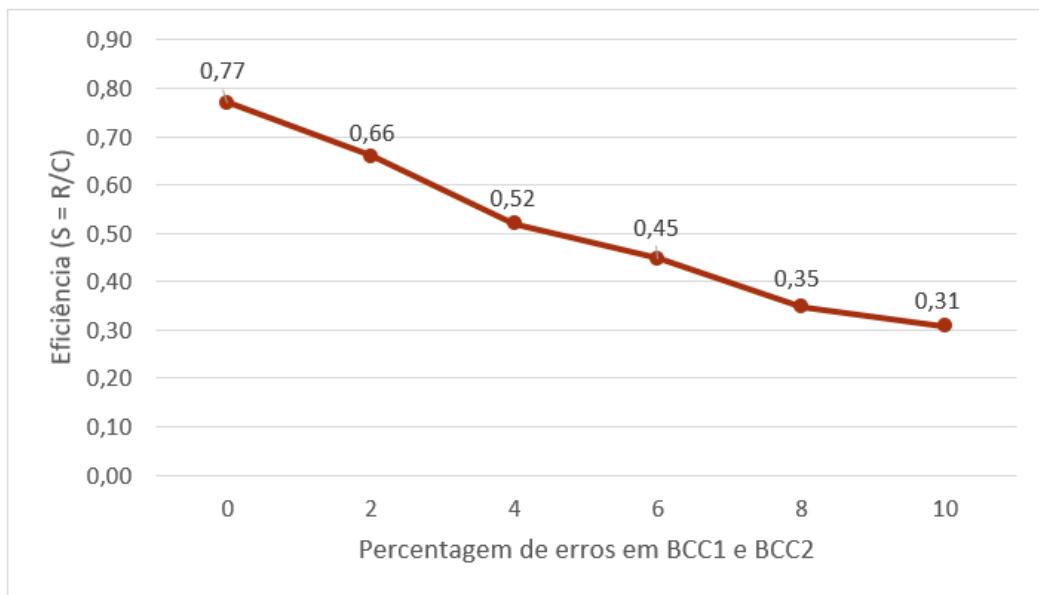
Os testes foram executados com um ficheiro de imagem com 130KB, de modo a que o grande número de tramas tornasse o efeito das variações testadas homogéneo em todos os ensaios.

Excetuando o teste à variação desse mesmo valor, o tamanho máximo de pacote de informação utilizado foi de 1024B.

As tabelas que contém os valores com os quais os gráficos foram gerados encontram-se no fim do relatório, nos anexos.

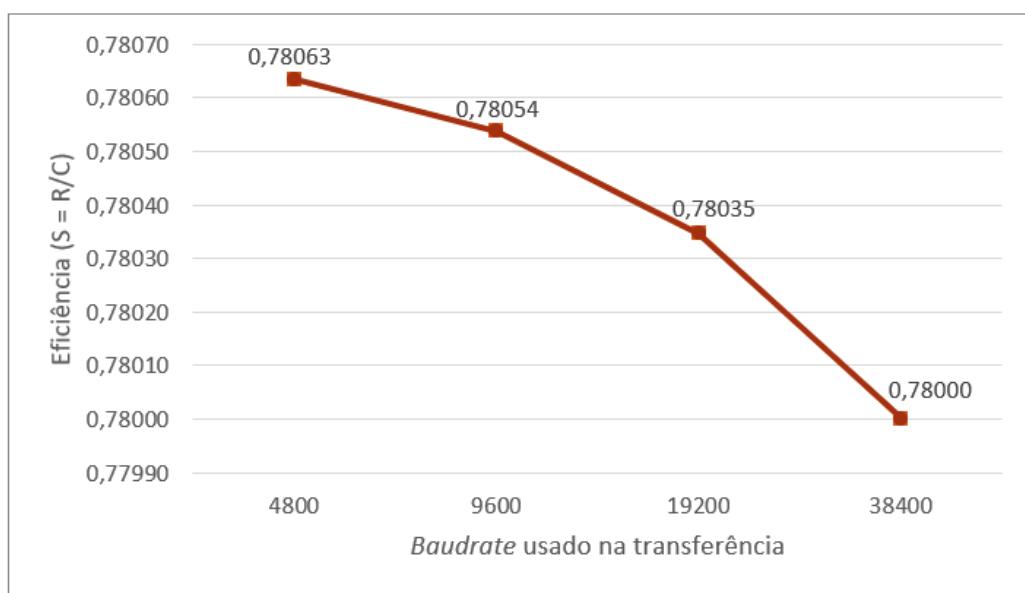
## Variação do FER

Com o gráfico do valor de S em função da percentagem de erros simulados, podemos concluir que o FER tem um impacto significativo na eficiência do programa. Isto deve-se, primariamente, ao facto de que, quando é gerado um erro no BCC1, irá ocorrer um *timeout*, que resultará na ausência de resposta por parte do receptor, por um número previamente definido de segundos (no nosso caso, **3 segundos**), o que afeta negativamente o tempo de execução. Já os erros no BCC2 não têm um impacto tão grande, pois apenas causam o reenvio da trama, que é imediato.



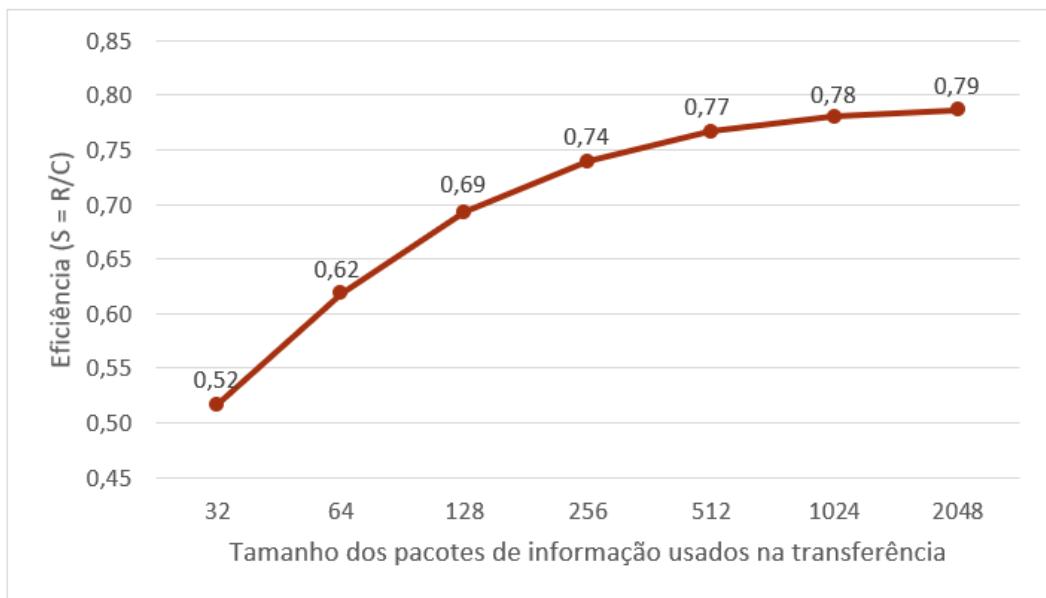
## Variação do baudrate (C – capacidade de ligação)

Com este gráfico, podemos concluir que, com o aumento da capacidade de ligação, diminui a eficiência.



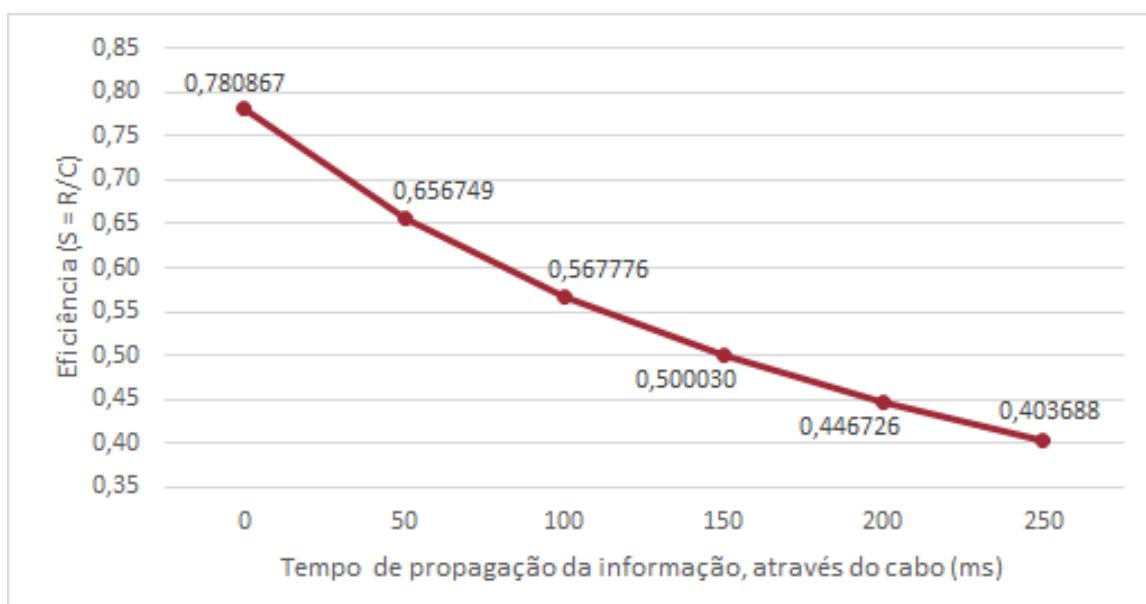
## Variação do tamanho das *I-frames*

Com o gráfico correspondente à variação de S, com base no tamanho das *I-frames*, mostramos que, quanto maior o tamanho de cada pacote de dados, mais eficiente será o programa. Isto verifica-se porque cada envio contém mais informação, o que reduz o número de tramas a enviar. Isto também leva a que o *timeout* ocorra menos vezes.



## Variação do tempo de propagação (T\_prop)

Como seria de esperar, com a variação do tempo de propagação de cada trama de informação, menos eficiente seria o programa. Isto deve-se naturalmente ao facto de a aplicação passar mais tempo sem enviar tramas uma vez que é simulado que o envio/receção de uma trama demora mais. Os valores presentes no gráfico foram calculados utilizando o envio do ficheiro pinguim.gif que foi disponibilizado. Utilizando a função *usleep()*, foi simulado um aumento no tempo de propagação, sendo que esse aumento vai desde 0 ms, até 250 ms.



Falando agora de um ponto de vista mais teórico sobre o protocolo *Stop & Wait*, após a transmissão de uma trama de informação, o emissor espera por uma confirmação positiva por parte do receptor, denominada por *acknowledgment*, ACK. Quando o receptor recebe a trama, caso esta não tenha erros, confirma então com ACK; caso contrário, é enviado um NACK (*negative acknowledgment*). Do lado do emissor, ao receber um ACK, este irá enviar a trama seguinte (se existir), mas se receber um NACK irá reenviar a mesma trama. De modo a prevenir o mau funcionamento do programa caso a trama I ou as respostas ACK ou NACK não sejam recebidas, o programa deve implementar um mecanismo de timeout, fazendo com que a trama I seja reenviada.

De modo ao receptor conseguir identificar se uma trama é nova ou se é duplicada, e também para o emissor saber qual a trama enviada a que um ACK se refere, tanto as tramas I como os ACKs devem ser numerados, com um 0 ou 1, sendo que estes valores são utilizados alternadamente (ACK(i) significa que o receptor está à espera de uma trama I(i)).

Na nossa aplicação foi então utilizado um protocolo *Stop & Wait* para o controlo de erros. As tramas I são identificadas com um 0 ou 1, sendo que a resposta do receptor pode ou ser RR (equivalente a ACK) ou REJ (equivalente a NACK). Estes últimos dois encontram-se devidamente identificadas com o seu número, 0 ou 1, dependendo do número da trama de informação recebida.

Trama enviada pelo emissor, e a resposta equivalente dada pelo receptor:

$N_s = 0 \Rightarrow$  Sem erros: RR ( $N_r = 1$ ); Com erros: REJ ( $N_r = 0$ )

$N_s = 1 \Rightarrow$  Sem erros: RR ( $N_r = 0$ ); Com erros: REJ ( $N_r = 1$ )

## **Conclusão**

### **Síntese**

Este projeto consiste no desenvolvimento de um serviço fiável de comunicação entre dois computadores ligados por porta de série, implementando o **protocolo de ligação de dados**.

### **Reflexão**

Este trabalho permitiu-nos compreender o protocolo de ligação de dados, incidindo sobre a estrutura das tramas e o processo de encapsulamento, envio e receção da informação.

Adicionalmente, é destacada a importância da **independência entre camadas**, sendo que cada camada do programa respeita esse conceito. A camada da ligação de dados não recorre, de todo, a qualquer processamento desenvolvido na camada da aplicação. Já a camada da aplicação não conhece quaisquer detalhes da implementação da camada da ligação de dados, conhecendo apenas como utilizar as suas funcionalidades.

### **Nota sobre o número de páginas utilizadas**

Embora tenhamos conhecimento que o limite pedido do número de páginas seja 8, ao adicionar algumas imagens e gráficos e ao espaçar certas partes do texto, o relatório acabou por ficar com um número de páginas ligeiramente acima do pedido, mas somos da opinião que desta maneira, a legibilidade e estruturação das várias partes do relatório ficam melhor. Pedimos a compreensão do professor relativamente a este tópico.

# Anexo - Código Fonte

alarm.c:

```
#include "alarm.h"

/**
 * Handles the alarm signal
 * @param signal Signal that is received
 */
void alarmHandler(int signal) {

    if(num_retr < ll.numTransmissions){
        resendFrame = true;
        printf("Timeout/invalid value: Sent frame again (numretries = %d)\n", num_retr);
        alarm(ll.timeout);
        num_retr++;
    }
    else{
        printf("Number of retries exceeded (numretries = %d)\n", num_retr);
        finish = 1;
    }
}
```

alarm.h:

```
#pragma once

#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <stdbool.h>
#include "data_link.h"
#include "macros.h"
#include "app.h"
#include "aux.h"

int finish, num_retr;
bool resendFrame;

/**
 * Handles the alarm signal
 * @param signal Signal that is received
 */
void alarmHandler(int signal);
```

## app.c:

```
#include "app.h"
#include "macros.h"
#include "aux.h"
#include "data_link.h"
#include "files.h"

/** 
 * Function that builds an application data packet, receiving a sequence number, and a data buffer
 * containing the bytes to be sent; returns the information on the packet buffer.
 * @param packetBuffer Buffer that will have the final contents of the packet
 * @param sequenceNumber Sequence number of the packet
 * @param dataBuffer Buffer with the data
 * @param dataLength Length of the data in the buffer
 * @return Length of the packet buffer
 */
int buildDataPacket(unsigned char *packetBuffer, int sequenceNumber, unsigned char *dataBuffer, int dataLength)
{
    packetBuffer[0] = CTRL_DATA;

    packetBuffer[1] = (unsigned char)sequenceNumber; // basta meter o int?

    int l1, l2;
    convertValueInTwo(dataLength, &l1, &l2);

    packetBuffer[2] = (unsigned char)l2;
    packetBuffer[3] = (unsigned char)l1;

    for (int i = 0; i < dataLength; i++)
        packetBuffer[i + 4] = dataBuffer[i];

    return dataLength + 4;
}

/** 
 * Function that builds a control packet
 * @param controlByte Can be CTRL_START or CTRL_END, to show if the control packet indicates the beginning or end of the file
 * @param packetBuffer Buffer that will have the final contents of the packet
 * @param fileSize Size of the full file, in bytes
 * @param fileName Name of the file
 * @return Length of the packet buffer
 */
int buildControlPacket(unsigned char controlByte, unsigned char *packetBuffer, int fileSize, char *fileName)
{
    packetBuffer[0] = controlByte;

    packetBuffer[1] = TYPE_FILESIZE;

    int length = 0;
    int currentFileSize = fileSize;

    // cicle to separate file size (v1) in bytes
    while (currentFileSize > 0)
    {
        int rest = currentFileSize % 256;
        int div = currentFileSize / 256;
        length++;

        // shifts all bytes to the right, to make space for the new byte
        for (unsigned int i = 2 + length; i > 3; i--)
            packetBuffer[i] = packetBuffer[i - 1];

        packetBuffer[3] = (unsigned char)rest;

        currentFileSize = div;
    }

    packetBuffer[2] = (unsigned char)length;

    packetBuffer[3 + length] = TYPE_FILENAME;

    int fileNameStart = 5 + length; // beginning of v2

    packetBuffer[4 + length] = (unsigned char)(strlen(fileName) + 1); // adds file name length (including '\0')

    for (unsigned int j = 0; j < (strlen(fileName) + 1); j++)
    { // strlen(fileName) + 1 in order to add the '\0' char
        packetBuffer[fileNameStart + j] = fileName[j];
    }

    return 3 + length + 2 + strlen(fileName) + 1; // total length of the packet
}
```

```

/***
 * Function, to be called by the reader, that parses the data packets
 * @param packetBuffer Buffer with the data packet
 * @param data Pointer to the file data packet extracted, to be returned by the function
 * @param sequenceNumber Pointer to the sequence number of the packet, to be returned by the function
 * @return 0 if it was sucessful; negative value otherwise
 */
int parseDataPacket(unsigned char *packetBuffer, unsigned char *data, int *sequenceNumber)
{
    if (packetBuffer[0] != CTRL_DATA)
    {
        return -1;
    }

    *sequenceNumber = (int)packetBuffer[1];

    int size_of_data;

    size_of_data = convertValueInOne((int)packetBuffer[3], (int)packetBuffer[2]);

    for (int i = 0; i < size_of_data; i++)
    {
        data[i] = packetBuffer[i + 4];
    }

    return 0;
}

/***
 * Function, to be called by the reader, that parses the control packets
 * @param packetBuffer Buffer with the control packet
 * @param fileSize Pointer to the size of the file, to be returned by the function
 * @param fileName Pointer to the name of the file, to be returned by the function
 * @return 0 if it was sucessful; negative value otherwise
 */
int parseControlPacket(unsigned char *packetBuffer, int *fileSize, char *fileName)
{
    if (packetBuffer[0] != CTRL_START && packetBuffer[0] != CTRL_END)
    {
        return -1;
    }

    int length1;
    if (packetBuffer[1] == TYPE_FILESIZE)
    {
        *fileSize = 0;
        length1 = (int)packetBuffer[2];

        for (int i = 0; i < length1; i++)
        {
            *fileSize = *fileSize * 256 + (int)packetBuffer[3 + i];
        }
    }
    else
    {
        return -1;
    }

    int length2;
    int fileNameStart = 5 + length1;

    if (packetBuffer[fileNameStart - 2] == TYPE_FILENAME)
    {
        length2 = (int)packetBuffer[fileNameStart - 1];

        for (int i = 0; i < length2; i++)
        {
            fileName[i] = packetBuffer[fileNameStart + i];
        }
    }
    else
    {
        return -1;
    }

    return 0;
}

```

```

/***
 * Function to receive a file, sent through the serial port
 * @param port Name of the serial port
 * @return 0 if successful; negative if an error occurs
 */
int receiveFile(char *port)
{
    // fills appLayer fields
    al.status = RECEIVER;

    if ((al.fileDescriptor = llopen(port, al.status)) <= 0)
    {
        return -1;
    }

    printf("\n-----llopen done-----\n\n");

    unsigned char packetBuffer[MAX_PACK_SIZE];

    int packetSize;
    int fileSize;
    unsigned char data[MAX_DATA_SIZE];
    char fileName[255];

    packetSize = llread(al.fileDescriptor, packetBuffer);

    if (packetSize < 0)
    {
        return -1;
    }

    // if start control packet was received
    if (packetBuffer[0] == CTRL_START)
    {
        if (parseControlPacket(packetBuffer, &fileSize, fileName) < 0)
        {
            return -1;
        }
    }
    else
    {
        return -1;
    }

    FILE *fp = openFile(fileName, "w");
    if (fp == NULL)
        return -1;

    int expectedSequenceNumber = 0;

    // starts received data packets (file data)
    while (1)
    {
        packetSize = llread(al.fileDescriptor, packetBuffer);

        if (packetSize < 0)
        {
            return -1;
        }

        // received data packet
        if (packetBuffer[0] == CTRL_DATA)
        {
            int sequenceNumber;

            if (parseDataPacket(packetBuffer, data, &sequenceNumber) < 0)
                return -1;

            // if sequence number doesn't match
            if (expectedSequenceNumber != sequenceNumber)
            {
                printf("Sequence number does not match!\n");
                return -1;
            }

            expectedSequenceNumber = (expectedSequenceNumber + 1) % 256;

            int dataLength = packetSize - 4;

            // writes to the file the content read from the serial port
            if (fwrite(data, sizeof(unsigned char), dataLength, fp) != dataLength)
            {
                return -1;
            }
        }
        // received end packet; file was fully transmitted
        else if (packetBuffer[0] == CTRL_END)
        {
            break;
        }
    }
}

```

```

    if (getFileSize(fp) != fileSize)
    {
        printf("file size does not match\n");
        return -1;
    }

    int fileSizeEnd;
    char fileNameEnd[255];

    if (parseControlPacket(packetBuffer, &fileSizeEnd, fileNameEnd) < 0)
    {
        return -1;
    }

    if((fileSize != fileSizeEnd) || (strcmp(fileNameEnd, fileName) != 0)){
        printf("Information in start and end packets does not match");
        return -1;
    }

    // close, in non canonical
    if (llclose(al.fileDescriptor, al.status) < 0)
    {
        return -1;
    }

    printf("\n-----llclose done-----\n\n");

    return 0;
}

/***
 * Function to send a file, using the serial port, to its destination
 * @param port Name of the serial port
 * @param fileName Name of the file to be sent
 * @return 0 if successful; negative if an error occurs
 */
int.sendFile(char *port, char *fileName) {

    FILE *fp = openFile(fileName, "r");
    if (fp == NULL){
        printf("Cannot find the file to transmit\n");
        return -1;
    }

    // fills appLayer fields
    al.status = TRANSMITTER;
    if ((al.fileDescriptor = llopen(port, al.status)) <= 0)
    {
        return -1;
    }

    printf("\n-----llopen done-----\n\n");

    unsigned char packetBuffer[MAX_PACK_SIZE];
    int fileSize = getFileSize(fp);

    int packetSize = buildControlPacket(CTRL_START, packetBuffer, fileSize, fileName);

    // sends control start packet, to indicate the start of the file transfer
    if (llwrite(al.fileDescriptor, packetBuffer, packetSize) < 0)
    {
        closeFile(fp);
        return -1;
    }

    printf("\n-----STARTING TO SEND FILE-----\n\n");

    unsigned char data[MAX_DATA_SIZE];
    int length_read;
    int sequenceNumber = 0;

    while (1)
    {
        // reads a data chunk from the file
        length_read = fread(data, sizeof(unsigned char), MAX_DATA_SIZE, fp);

        if (length_read != MAX_DATA_SIZE)
        {
            if (feof(fp))
            {

                packetSize = buildDataPacket(packetBuffer, sequenceNumber, data, length_read);
                sequenceNumber = (sequenceNumber + 1) % 256;

                // sends the last data frame
                if (llwrite(al.fileDescriptor, packetBuffer, packetSize) < 0)
                {
                    closeFile(fp);
                    return -1;
                }
            }
        }
    }
}

```

```

        break;
    }
    else
    {
        perror("error reading file data");
        return -1;
    }
}

packetSize = buildDataPacket(packetBuffer, sequenceNumber, data, length_read);
sequenceNumber = (sequenceNumber + 1) % 256;

// sends a data frame
if (llwrite(al.fileDescriptor, packetBuffer, packetSize) < 0)
{
    closeFile(fp);
    return -1;
}

}

packetSize = buildControlPacket(CTRL_END, packetBuffer, fileSize, fileName);

// sends control end packet; indicating the end of the file transfer
if (llwrite(al.fileDescriptor, packetBuffer, packetSize) < 0)
{
    closeFile(fp);
    return -1;
}

printf("\n-----ENDED SENDING FILE-----\n\n");

if (llclose(al.fileDescriptor, al.status) < 0)
    return -1;

printf("\n-----llclose done-----\n\n");

if (closeFile(fp) != 0)
    return -1;

return 0;
}

```

## app.h:

```

#pragma once

#include <stdio.h>

struct applicationLayer {
    int fileDescriptor; /*Descriptor correspondente à porta série*/
    int status; /*TRANSMITTER | RECEIVER*/
};

struct applicationLayer al;

/** 
 * Function that builds an application data packet, receiving a sequence number, and a data buffer
 * containing the bytes to be sent; returns the information on the packet buffer.
 * @param packetBuffer Buffer that will have the final contents of the packet
 * @param sequenceNumber Sequence number of the packet
 * @param dataBuffer Buffer with the data
 * @param dataLength Length of the data in the buffer
 * @return Length of the packet buffer
 */
int buildDataPacket(unsigned char* packetBuffer, int sequenceNumber, unsigned char* dataBuffer, int dataLength);

/** 
 * Function that builds a control packet
 * @param controlByte Can be CTRL_START or CTRL_END, to show if the control packet indicates the beginning or end of the file
 * @param packetBuffer Buffer that will have the final contents of the packet
 * @param fileSize Size of the full file, in bytes
 * @param fileName Name of the file
 * @return Length of the packet buffer
 */
int buildControlPacket(unsigned char controlByte, unsigned char* packetBuffer, int fileSize, char* fileName);

/** 
 * Function, to be called by the reader, that parses the control packets
 * @param packetBuffer Buffer with the control packet
 * @param fileSize Pointer to the size of the file, to be returned by the function
 * @param fileName Pointer to the name of the file, to be returned by the function
 * @return 0 if it was sucessful; negative value otherwise
 */
int parseControlPacket(unsigned char* packetBuffer, int* fileSize, char* fileName);

/** 
 * Function, to be called by the reader, that parses the data packets
 * @param packetBuffer Buffer with the data packet
 * @param data Pointer to the file data packet extracted, to be returned by the function
 */

```

```

 * @param sequenceNumber Pointer to the sequence number of the packet, to be returned by the function
 * @return 0 if it was sucessful; negative value otherwise
 */
int parseDataPacket(unsigned char* packetBuffer, unsigned char* data, int* sequenceNumber);

/**
 * Function to send a file, using the serial port, to its destination
 * @param port Name of the serial port
 * @param fileName Name of the file to be sent
 * @return 0 if successful; negative if an error occurs
 */
int.sendFile(char *port , char* fileName);

/**
 * Function to receive a file, sent through the serial port
 * @param port Name of the serial port
 * @return 0 if successful; negative if an error occurs
 */
int.receiveFile(char *port);

```

## AUX.C:

```

#include <termios.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>
#include <unistd.h>
#include <fcntl.h>
#include "app.h"
#include "macros.h"
#include "statemachine.h"
#include "data_link.h"
#include "alarm.h"

/**
 * Function to create the Block Check Character relative to the Address and Control fields
 * @param a Address Character of the frame
 * @param c Control Character of the frame
 * @return Expected value for the Block Check Character
 */
unsigned char createBCC(unsigned char a, unsigned char c) {
    return a ^ c;
}

/**
 * Function to create the Block Check Character relative to the Data Characters of the frame
 * @param frame Frame position where the Data starts
 * @param length Number of Data Characters to process
 * @return Expected value for the Block Check Character
 */
unsigned char.createBCC_2(unsigned char* frame, int length) {

    unsigned char bcc2 = frame[0];

    for(int i = 1; i < length; i++){
        bcc2 = bcc2 ^ frame[i];
    }

    return bcc2;
}

/**
 * Function to apply byte stuffing to the Data Characters of a frame
 * @param frame Address of the frame
 * @param length Number of Data Characters to process

```

```

< @return Length of the new frame, post byte stuffing
*/
int byteStuffing(unsigned char* frame, int length) {

    // allocates space for auxiliary buffer (length of the packet, plus 6 bytes for the frame header and tail)
    unsigned char aux[length + 6];

    for(int i = 0; i < length + 6 ; i++){
        | aux[i] = frame[i];
    }

    // passes information from the frame to aux

    int finalLength = DATA_START;
    // parses aux buffer, and fills in correctly the frame buffer
    for(int i = DATA_START; i < (length + 6); i++){

        if(aux[i] == FLAG && i != (length + 5)) {
            | frame[finalLength] = ESCAPE_BYTE;
            | frame[finalLength+1] = BYTE_STUFFING_FLAG;
            finalLength = finalLength + 2;
        }
        else if(aux[i] == ESCAPE_BYTE && i != (length + 5)) {
            | frame[finalLength] = ESCAPE_BYTE;
            | frame[finalLength+1] = BYTE_STUFFING_ESCAPE;
            finalLength = finalLength + 2;
        }
        else{
            | frame[finalLength] = aux[i];
            finalLength++;
        }
    }

    return finalLength;
}

/***
 * Function to reverse the byte stuffing applied to the Data Characters of a frame
 * @param frame Address of the frame
 * @param length Number of Data Characters to process
 * @return Length of the new frame, post byte destuffing
 */
int byteDestuffing(unsigned char* frame, int length) {

    // allocates space for the maximum possible frame length read (length of the data packet + bcc2, already with stuffing,
    // plus the other 5 bytes in the frame)
    unsigned char aux[length + 5];

    // copies the content of the frame (with stuffing) to the aux frame
    for(int i = 0; i < (length + 5) ; i++) {
        | aux[i] = frame[i];
    }

    int finalLength = DATA_START;

    // iterates through the aux buffer, and fills the frame buffer with destuffed content
    for(int i = DATA_START; i < (length + 5); i++) {

        if(aux[i] == ESCAPE_BYTE){
            if (aux[i+1] == BYTE_STUFFING_ESCAPE) {
                | frame[finalLength] = ESCAPE_BYTE;
            }
            else if(aux[i+1] == BYTE_STUFFING_FLAG) {
                | frame[finalLength] = FLAG;
            }
            i++;
            finalLength++;
        }
        else{
            | frame[finalLength] = aux[i];
            finalLength++;
        }
    }

    return finalLength;
}

/***
 * Function to create a supervision frame for the serial port file transfer protocol
 * @param frame Address where the frame will be stored
 * @param controlField Control field of the supervision frame
 * @param role Role for which to create the frame, marking the difference between the Transmitter and the Receiver
 * @return 0 if successful; negative if an error occurs
 */
int createSupervisionFrame(unsigned char* frame, unsigned char controlField, int role) {

    | frame[0] = FLAG;

```

```

    if(role == TRANSMITTER) {
        if(controlField == SET || controlField == DISC) {
            ||| frame[1] = END_SEND;
        }
        else if(controlField == UA || controlField == RR_0 || controlField == REJ_0 || controlField == RR_1 || controlField == REJ_1 ) {
            ||| frame[1] = END_REC;
        }
        else return -1;
    }
    else if(role == RECEIVER) {
        if(controlField == SET || controlField == DISC) {
            ||| frame[1] = END_REC;
        }
        else if(controlField == UA || controlField == RR_0 || controlField == REJ_0 || controlField == RR_1 || controlField == REJ_1 ) {
            ||| frame[1] = END_SEND;
        }
        else return -1;
    }
    else return -1;
}

frame[2] = controlField;

frame[3] = createBCC(frame[1], frame[2]);

frame[4] = FLAG;

return 0;
}

/***
 * Function to create an information frame for the serial port file transfer protocol
 * @param frame Address where the frame will be stored
 * @param controlField Control field of the supervision frame
 * @param infoField Start address of the information to be inserted into the information frame
 * @param infoFieldLength Number of data characters to be inserted into the information frame
 * @return Returns 0, as there is no place at which an error can occur
 */
int createInformationFrame(unsigned char* frame, unsigned char controlField, unsigned char* infoField, int infoFieldLength) {

    frame[0] = FLAG;

    frame[1] = END_SEND; // como so o emissor envia tramas I, assume-se que o campo de endereco e sempre 0x03.

    frame[2] = controlField;

    frame[3] = createBCC(frame[1], frame[2]);

    for(int i = 0; i < infoFieldLength; i++) {
        ||| frame[i + 4] = infoField[i];
    }

    unsigned bcc2 = createBCC_2(infoField, infoFieldLength);

    frame[infoFieldLength + 4] = bcc2;

    frame[infoFieldLength + 5] = FLAG;

    return 0;
}

/***
 * Function to send a frame to the designated file descriptor
 * @param frame Start address of the frame to be sent
 * @param fd File descriptor to which to write the information
 * @param length Size of the frame to be sent (size of information to be written)
 * @return Number of bytes written if successful; negative if an error occurs
 */
int sendFrame(unsigned char* frame, int fd, int length) {

    int n;

    if( (n = write(fd, frame, length)) <= 0){
        ||| return -1;
    }

    return n;
}

/***
 * Function to read a byte from the designated file descriptor
 * @param byte Address to which to store the byte
 * @param fd File descriptor from which to read the byte
 * @return Return value of the read() call if successful; negative if an error occurs
 */
int readByte(unsigned char* byte, int fd) {

    if(read(fd, byte, sizeof(unsigned char)) <= 0)
        ||| return -1;
}

```

```

    | return 0;
}

/***
 * Function to read a supervision frame, sent according to the serial port file transfer protocol
 * @param frame Address where the frame will be stored
 * @param fd File descriptor from which to read the frame
 * @param wantedBytes Array containing the possible expected control bytes of the frame
 * @param wantedBytesLength Number of possible expected control bytes of the frame
 * @param addressByte Address from which a frame is expected
 * @return Index of the wanted byte found, in the wantedBytes array
 */
int readSupervisionFrame(unsigned char* frame, int fd, unsigned char* wantedBytes, int wantedBytesLength, unsigned char addressByte) {

    state_machine_st *st = create_state_machine(wantedBytes, wantedBytesLength, addressByte);

    unsigned char byte;

    while(st->state != STOP && finish != 1 && !resendFrame) {
        if(readByte(&byte, fd) == 0)
            event_handler(st, byte, frame, SUPERVISION);
    }

    int ret = st->foundIndex;

    destroy_st(st);

    if(finish == 1 || resendFrame)
        return -1;

    return ret;
}

/***
 * Function to read an information frame, sent according to the serial port file transfer protocol
 * @param frame Address where the frame will be stored
 * @param fd File descriptor from which to read the frame
 * @param wantedBytes Array containing the possible expected control bytes of the frame
 * @param wantedBytesLength Number of possible expected control bytes of the frame
 * @param addressByte Address from which a frame is expected
 * @return Length of the data packet sent, including byte stuffing and BCC2
 */
int readInformationFrame(unsigned char* frame, int fd, unsigned char* wantedBytes, int wantedBytesLength, unsigned char addressByte) {

    state_machine_st *st = create_state_machine(wantedBytes, wantedBytesLength, addressByte);

    unsigned char byte;

    while(st->state != STOP) {
        if(readByte(&byte, fd) == 0)
            event_handler(st, byte, frame, INFORMATION);
    }

    // dataLength = length of the data packet sent from the application on the transmitter side
    //           (includes data packet + bcc2, with stuffing)
    int ret = st->dataLength;

    destroy_st(st);

    return ret;
}

/***
 * Function to open the file descriptor through which to execute the serial port communications,
 * in the non-canonical mode, according to the serial port file transfer protocol
 * @param port Name of the port to be opened
 * @param oldtio Struct where the pre-open port settings will be stored
 * @param vtime Value to be assigned to the VTIME field of the new settings - time between bytes read
 * @param vmin Value to be assigned to the VMIN field of the new settings - minimum amount of bytes to read
 * @return File descriptor that was opened with the given port
 */
int openNonCanonical(char* port, struct termios* oldtio, int vtime, int vmin) {

    int fd = open(port, O_RDWR | O_NOCTTY );
    if (fd <0) {
        perror(port);
        return -1;
    }

    struct termios newtio;

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

```

```

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

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

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

tcflush(fd, TCOFLUSH);

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

return fd;
}

/***
 * Function to close the file descriptor through which the serial port communications were executed
 * @param fd File descriptor where the port has been opened
 * @param oldtio Struct containing the original port settings have been saved, so they can be restored
 * @return 0 if successful; negative if an error occurs
 */
int closeNonCanonical(int fd, struct termios* oldtio) {

sleep(1);

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

return 0;
}

/***
 * Function to install the alarm handler, using sigaction
 */
void alarmHandlerInstaller() {
    struct sigaction action;
    action.sa_handler = alarmHandler;

    if(sigemptyset(&action.sa_mask) == -1){
        perror("sigemptyset");
        exit(-1);
    }

    action.sa_flags = 0;

    if(sigaction(SIGALRM, &action, NULL) != 0){
        perror("sigaction");
        exit(-1);
    }
}

// ----

/***
 * Auxiliary function to convert a decimal value into two (max. 8 bits) values, for hexadecimal representation
 * @param k Decimal value to be converted
 * @param l1 Least significant bits of the converted value
 * @param l2 Most significant bits of the converted value
 */
void convertValueInTwo(int k, int* l1, int* l2) {
    *l1 = k % 256;
    *l2 = k / 256;
}

/***
 * Auxiliary function to convert two (max. 8 bits) values, from hexadecimal representation, into one single decimal
 * @param l1 Least significant bits of the value to be converted
 * @param l2 Most significant bits of the value to be converted
 * @return Decimal converted value
 */
int convertValueInOne(int l1, int l2) {
    return 256 * l2 + l1;
}

```

## aux.h:

```
#pragma once

#include <termios.h>
#include <unistd.h>

/***
 * Function to create the Block Check Character relative to the Address and Control fields
 * @param a Address Character of the frame
 * @param c Control Character of the frame
 * @return Expected value for the Block Check Character
 */
unsigned char createBCC(unsigned char a, unsigned char c);

/***
 * Function to create the Block Check Character relative to the Data Characters of the frame
 * @param frame Frame position where the Data starts
 * @param length Number of Data Characters to process
 * @return Expected value for the Block Check Character
 */
unsigned char createBCC_2(unsigned char* frame, int length);

/***
 * Function to apply byte stuffing to the Data Characters of a frame
 * @param frame Address of the frame
 * @param length Number of Data Characters to process
 * @return Length of the new frame, post byte stuffing
 */
int byteStuffing(unsigned char* frame, int length);

/***
 * Function to reverse the byte stuffing applied to the Data Characters of a frame
 * @param frame Address of the frame
 * @param length Number of Data Characters to process
 * @return Length of the new frame, post byte destuffing
 */
int byteDestuffing(unsigned char* frame, int length);

/***
 * Function to create a supervision frame for the serial port file transfer protocol
 * @param frame Address where the frame will be stored
 * @param controlField Control field of the supervision frame
 * @param role Role for which to create the frame, marking the difference between the Transmitter and the Receiver
 * @return 0 if successful; negative if an error occurs
 */
int createSupervisionFrame(unsigned char* frame, unsigned char controlField, int role);

/***
 * Function to create an information frame for the serial port file transfer protocol
 * @param frame Address where the frame will be stored
 * @param controlField Control field of the supervision frame
 * @param infoField Start address of the information to be inserted into the information frame
 * @param infoFieldLength Number of data characters to be inserted into the information frame
 * @return Returns 0, as there is no place at which an error can occur
 */
int createInformationFrame(unsigned char* frame, unsigned char controlField, unsigned char* infoField, int infoFieldLength);

/***
 * Function to read a supervision frame, sent according to the serial port file transfer protocol
 * @param frame Address where the frame will be stored
 * @param fd File descriptor from which to read the frame
 * @param wantedBytes Array containing the possible expected control bytes of the frame
 * @param wantedBytesLength Number of possible expected control bytes of the frame
 * @param addressByte Address from which a frame is expected
 * @return Index of the wanted byte found, in the wantedBytes array
 */
int readSupervisionFrame(unsigned char* frame, int fd, unsigned char* wantedBytes, int wantedBytesLength, unsigned char addressByte);

/***
 * Function to read an information frame, sent according to the serial port file transfer protocol
 * @param frame Address where the frame will be stored
 * @param fd File descriptor from which to read the frame
 * @param wantedBytes Array containing the possible expected control bytes of the frame
 * @param wantedBytesLength Number of possible expected control bytes of the frame
 * @param addressByte Address from which a frame is expected
 * @return Length of the data packet sent, including byte stuffing and BCC2
 */
int readInformationFrame(unsigned char* frame, int fd, unsigned char* wantedBytes, int wantedBytesLength, unsigned char addressByte);
```

```

/**
 * Function to send a frame to the designated file descriptor
 * @param frame Start address of the frame to be sent
 * @param fd File descriptor to which to write the information
 * @param length Size of the frame to be sent (size of information to be written)
 * @return Number of bytes written if successful; negative if an error occurs
 */
int sendFrame(unsigned char* frame, int fd, int length);

/**
 * Function to read a byte from the designated file descriptor
 * @param byte Address to which to store the byte
 * @param fd File descriptor from which to read the byte
 * @return Return value of the read() call if successful; negative if an error occurs
 */
int readByte(unsigned char* byte, int fd);

/**
 * Function to open the file descriptor through which to execute the serial port communications,
 * in the non-canonical mode, according to the serial port file transfer protocol
 * @param port Name of the port to be opened
 * @param oldtio Struct where the pre-open port settings will be stored
 * @param vtime Value to be assigned to the VTIME field of the new settings - time between bytes read
 * @param vmin Value to be assigned to the VMIN field of the new settings - minimum amount of bytes to read
 * @return File descriptor that was opened with the given port
 */
int openNonCanonical(char* port, struct termios* oldtio, int vtime, int vmin);

/**
 * Function to close the file descriptor through which the serial port communications were executed
 * @param fd File descriptor where the port has been opened
 * @param oldtio Struct containing the original port settings have been saved, so they can be restored
 * @return 0 if successful; negative if an error occurs
 */
int closeNonCanonical(int fd, struct termios* oldtio);

/**
 * Function to install the alarm handler, using sigaction
 */
void alarmHandlerInstaller();

// ----

/**
 * Auxiliary function to convert a decimal value into two (max. 8 bits) values, for hexadecimal representation
 * @param k Decimal value to be converted
 * @param l1 Least significant bits of the converted value
 * @param l2 Most significant bits of the converted value
 */
void convertValueInTwo(int k, int* l1, int* l2);

/**
 * Auxiliary function to convert two (max. 8 bits) values, from hexadecimal representation, into one single decimal
 * @param l1 Least significant bits of the value to be converted
 * @param l2 Most significant bits of the value to be converted
 * @return Decimal converted value
 */
int convertValueInOne(int l1, int l2);

```

## data\_link.c:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>
#include <unistd.h>
#include <signal.h>
#include "data_link.h"
#include "statemachine.h"
#include "aux.h"
#include "app.h"
#include "alarm.h"

/***
 * Opens the connection for the receiver
 * @param fd File descriptor for the serial port
 * @return File descriptor; -1 in case of error
 */
int llOpenReceiver(int fd)
{
    unsigned char wantedByte[1];
    wantedByte[0] = SET;
    if (readSupervisionFrame(ll.frame, fd, wantedByte, 1, END_SEND) == -1)
        return -1;

    printf("Received SET frame\n");

    if (createSupervisionFrame(ll.frame, UA, RECEIVER) != 0)
        return -1;

    ll.frameLength = BUF_SIZE_SUP;

    // send SET frame to receiver
    if (sendFrame(ll.frame, fd, ll.frameLength) == -1)
        return -1;

    printf("Sent UA frame\n");

    return fd;
}

/***
 * Opens the connection for the transmitter
 * @param fd File descriptor for the serial port
 * @return File descriptor; -1 in case of error
 */
int llOpenTransmitter(int fd)
{
    unsigned char responseBuffer[BUF_SIZE_SUP]; // buffer to read the response

    // creates SET frame
    if (createSupervisionFrame(ll.frame, SET, TRANSMITTER) != 0)
        return -1;

    ll.frameLength = BUF_SIZE_SUP;

    // send SET frame to receiver
    if (sendFrame(ll.frame, fd, ll.frameLength) == -1)
        return -1;

    printf("Sent SET frame\n");

    int read_value = -1;
    finish = 0;
    num_retr = 0;
    resendFrame = false;

    alarm(ll.timeout);

    unsigned char wantedByte[1];
    wantedByte[0] = UA;

    while (finish != 1)
    {
        read_value = readSupervisionFrame(responseBuffer, fd, wantedByte, 1, END_SEND);
        if (resendFrame)
        {
            sendFrame(ll.frame, fd, ll.frameLength);
            resendFrame = false;
        }
    }
}
```

```

        if (read_value >= 0)
        {
            // Cancels alarm
            alarm(0);
            finish = 1;
        }
    }

    if (read_value == -1)
    {
        printf("Closing file descriptor\n");
        return -1;
    }

    printf("Received UA frame\n");

    return fd;
}

/***
 * Function that opens and establishes the connection between the receiver and the transmitter
 * @param port Port name
 * @param role Flag that indicates the transmitter or the receiver
 * @return File descriptor; -1 in case of error
 */
int llopen(char *port, int role)
{
    strcpy(ll.port, port);
    ll.baudRate = BAUDRATE;
    ll.numTransmissions = NUM_RETRY;
    ll.timeout = TIMEOUT;
    ll.sequenceNumber = 0;

    int fd;

    // open, in non canonical
    if ((fd = openNonCanonical(port, &oldtio, VTIME_VALUE, VMIN_VALUE)) == -1)
        return -1;

    // installs alarm handler
    alarmHandlerInstaller();

    int returnFd;

    if (role == TRANSMITTER)
    {
        returnFd = llOpenTransmitter(fd);
        if (returnFd < 0)
        {
            closeNonCanonical(fd, &oldtio);
            return -1;
        }
        else
            return returnFd;
    }
    else if (role == RECEIVER)
    {
        returnFd = llOpenReceiver(fd);
        if (returnFd < 0)
        {
            closeNonCanonical(fd, &oldtio);
            return -1;
        }
        else
            return returnFd;
    }

    perror("Invalid role");
    closeNonCanonical(fd, &oldtio);
    return -1;
}

/***
 * Function that writes the information contained in the buffer to the serial port
 * @param fd File descriptor of the serial port
 * @param buffer Information to be written
 * @param length Length of the buffer
 * @return Number of characters written; -1 in case of error
 */
int llwrite(int fd, unsigned char *buffer, int length)
{
    unsigned char responseBuffer[BUFSIZE]; // buffer to receive the response

    unsigned char controlByte;
    if (ll.sequenceNumber == 0)
        controlByte = S_0;
    else
        controlByte = S_1;
}

```

```

if (createInformationFrame(ll.frame, controlByte, buffer, length) != 0)
{
    closeNonCanonical(fd, &oldtio);
    return -1;
}

int fullLength; // frame length after stuffing

if ((fullLength = byteStuffing(ll.frame, length)) < 0)
{
    closeNonCanonical(fd, &oldtio);
    return -1;
}

ll.frameLength = fullLength;
int numWritten;

bool dataSent = false;

while (!dataSent)
{
    if ((numWritten = sendFrame(ll.frame, fd, ll.frameLength)) == -1)
    {
        closeNonCanonical(fd, &oldtio);
        return -1;
    }

    printf("Sent I frame\n");

    int read_value = -1;
    finish = 0;
    num_retr = 0;
    resendFrame = false;

    alarm(ll.timeout);

    unsigned char wantedBytes[2];

    if (controlByte == S_0)
    {
        wantedBytes[0] = RR_1;
        wantedBytes[1] = REJ_0;
    }
    else if (controlByte == S_1)
    {
        wantedBytes[0] = RR_0;
        wantedBytes[1] = REJ_1;
    }

    while (finish != 1)
    {
        read_value = readSupervisionFrame(responseBuffer, fd, wantedBytes, 2, END_SEND);

        if (resendFrame)
        {
            sendFrame(ll.frame, fd, ll.frameLength);
            resendFrame = false;
        }

        if (read_value >= 0)
        { // read_value é o indice do wantedByte que foi encontrado
            // Cancels alarm
            alarm(0);
            finish = 1;
        }
    }

    if (read_value == -1)
    {
        printf("Closing file descriptor\n");

        closeNonCanonical(fd, &oldtio);
        return -1;
    }

    if (read_value == 0) // read a RR
        dataSent = true;
    else // read a REJ
        dataSent = false;

    printf("Received response frame (%x)\n", responseBuffer[2]);
}

if (ll.sequenceNumber == 0)
    ll.sequenceNumber = 1;
else if (ll.sequenceNumber == 1)
    ll.sequenceNumber = 0;
else
    return -1;

return (numWritten - 6); // length of the data packet length sent to the receiver
}

```

```

/**
 * Function that reads the information written in the serial port
 * @param fd File descriptor of the serial port
 * @param buffer Array of characters where the read information will be stored
 * @return Number of characters read; -1 in case of error
 */
int llread(int fd, unsigned char *buffer)
{

    int numBytes;
    unsigned char wantedBytes[2];
    wantedBytes[0] = S_0;
    wantedBytes[1] = S_1;

    int read_value;

    bool isBufferFull = false;

    while (!isBufferFull)
    {

        read_value = readInformationFrame(ll.frame, fd, wantedBytes, 2, END_SEND);

        printf("Received I frame\n");

        if ((numBytes = byteDestuffing(ll.frame, read_value)) < 0)
        {
            closeNonCanonical(fd, &oldtio);
            return -1;
        }

        int controlByteRead;
        if (ll.frame[2] == S_0)
            controlByteRead = 0;
        else if (ll.frame[2] == S_1)
            controlByteRead = 1;

        unsigned char responseByte;
        if (ll.frame[numBytes - 2] == createBCC_2(&ll.frame[DATA_START], numBytes - 6))
        { // if bcc2 is correct

            if (controlByteRead != ll.sequenceNumber)
            { // duplicated trama; discard information

                // ignora dados da trama
                if (controlByteRead == 0)
                {
                    responseByte = RR_1;
                    ll.sequenceNumber = 1;
                }
                else
                {
                    responseByte = RR_0;
                    ll.sequenceNumber = 0;
                }
            }
            else
            { // new trama

                // passes information to the buffer
                for (int i = 0; i < numBytes - 6; i++)
                {
                    buffer[i] = ll.frame[DATA_START + i];
                }

                isBufferFull = true;

                if (controlByteRead == 0)
                {
                    responseByte = RR_1;
                    ll.sequenceNumber = 1;
                }
                else
                {
                    responseByte = RR_0;
                    ll.sequenceNumber = 0;
                }
            }
        }
        else
        { // if bcc2 is not correct
            if (controlByteRead != ll.sequenceNumber)
            { // duplicated trama

                // ignores frame data

                if (controlByteRead == 0)
                {

```

```

        responseByte = RR_1;
        ll.sequenceNumber = 1;
    }
    else
    {
        responseByte = RR_0;
        ll.sequenceNumber = 0;
    }
}
else
{ // new trama

    // ignores frame data, because of error

    if (controlByteRead == 0)
    {
        responseByte = REJ_0;
        ll.sequenceNumber = 0;
    }
    else
    {
        responseByte = REJ_1;
        ll.sequenceNumber = 1;
    }
}

if (createSupervisionFrame(ll.frame, responseByte, RECEIVER) != 0)
{
    closeNonCanonical(fd, &oldtio);
    return -1;
}

ll.frameLength = BUF_SIZE_SUP;

// send RR/REJ frame to receiver
if (sendFrame(ll.frame, fd, ll.frameLength) == -1)
{
    closeNonCanonical(fd, &oldtio);
    return -1;
}

printf("Sent response frame (%x)\n", ll.frame[2]);
}

return (numBytes - 6); // number of bytes of the data packet read
}

/***
 * Closes the connection for the transmitter
 * @param fd File descriptor for the serial port
 * @return Positive value when sucess; negative value when error
 */
int llCloseTransmitter(int fd)
{
    unsigned char responseBuffer[BUF_SIZE_SUP]; // buffer to receive the response

    ll.frameLength = BUF_SIZE_SUP;

    // creates DISC frame
    if (createSupervisionFrame(ll.frame, DISC, TRANSMITTER) != 0)
        return -1;

    // send DISC frame to receiver
    if (sendFrame(ll.frame, fd, ll.frameLength) == -1)
        return -1;

    printf("Sent DISC frame\n");

    int read_value = -1;
    finish = 0;
    num_retr = 0;
    resendFrame = false;

    alarm(ll.timeout);

    unsigned char wantedByte[1];
    wantedByte[0] = DISC;

    while (finish != 1)
    {
        read_value = readSupervisionFrame(responseBuffer, fd, wantedByte, 1, END_REC);

        if (resendFrame)
        {
            sendFrame(ll.frame, fd, ll.frameLength);
            resendFrame = false;
        }
    }
}

```

```

    if (read_value >= 0)
    {
        // Cancels alarm
        alarm(0);
        finish = 1;
    }
}

if (read_value == -1)
{
    printf("Closing file descriptor\n");
    return -1;
}

printf("Received DISC frame\n");

// creates UA frame
if (createSupervisionFrame(ll.frame, UA, TRANSMITTER) != 0)
    return -1;

// send DISC frame to receiver
if (sendFrame(ll.frame, fd, ll.frameLength) == -1)
    return -1;

printf("Sent UA frame\n");

return 0;
}

/***
 * Closes the connection for the receiver
 * @param fd File descriptor for the serial port
 * @return Positive value when sucess; negative value when error
 */
int llCloseReceiver(int fd)
{
    unsigned char responseBuffer[BUF_SIZE_SUP]; // buffer to receive the response

    ll.frameLength = BUF_SIZE_SUP;

    unsigned char wantedByte[1];
    wantedByte[0] = DISC;

    if (readSupervisionFrame(ll.frame, fd, wantedByte, 1, END_SEND) == -1)
        return -1;

    printf("Received DISC frame\n");

    // creates DISC frame
    if (createSupervisionFrame(ll.frame, DISC, RECEIVER) != 0)
        return -1;

    // send DISC frame to receiver
    if (sendFrame(ll.frame, fd, ll.frameLength) == -1)
        return -1;

    printf("Sent DISC frame\n");

    int read_value = -1;
    finish = 0;
    num_retr = 0;
    resendFrame = false;

    alarm(ll.timeout);

    wantedByte[0] = UA;

    while (finish != 1)
    {
        read_value = readSupervisionFrame(responseBuffer, fd, wantedByte, 1, END_REC);

        if (resendFrame)
        {
            sendFrame(ll.frame, fd, ll.frameLength);
            resendFrame = false;
        }

        if (read_value >= 0)
        {
            // Cancels alarm
            alarm(0);
            finish = 1;
        }
    }

    if (read_value == -1)
    {
        printf("Closing file descriptor\n");
        return -1;
    }
}

```

```

    printf("Received UA frame\n");

    return 0;
}

/***
 * Function that closes the connection between the receiver and the transmitter
 * @param fd File descriptor of the port
 * @param role Flag that indicates the transmitter or the receiver
 * @return Positive value when sucess; negative value when error
 */
int llclose(int fd, int role)
{
    if (role == TRANSMITTER)
    {
        if (llCloseTransmitter(fd) < 0)
        {
            closeNonCanonical(fd, &oldtio);
            return -1;
        }
    }
    else if (role == RECEIVER)
    {
        if (llCloseReceiver(fd) < 0)
        {
            closeNonCanonical(fd, &oldtio);
            return -1;
        }
    }
    else
    {
        perror("Invalid role");
        return -1;
    }

    // close, in non canonical
    if (closeNonCanonical(fd, &oldtio) == -1)
        return -1;

    if (close(fd) != 0)
        return -1;

    return 1;
}

```

## data\_link.h:

```

#pragma once

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

struct linkLayer {
    char port[20]; /*Dispositivo /dev/ttySx, x = 0, 1*/
    int baudRate; /*Velocidade de transmissão*/
    unsigned int sequenceNumber; /*Número de sequência da trama: 0, 1*/
    unsigned int timeout; /*Valor do temporizador: 1 s*/
    unsigned int numTransmissions; /*Número de tentativas em caso de falha*/
    unsigned char frame[MAX_SIZE_FRAME]; /*Trama*/
    unsigned int frameLength; /*Comprimento atual da trama*/
};

// global variables
struct linkLayer ll;
struct termios oldtio;

/***
 * Opens the connection for the receiver
 * @param fd File descriptor for the serial port
 * @return File descriptor; -1 in case of error
 */
int llOpenReceiver(int fd);

/***
 * Opens the connection for the transmitter
 * @param fd File descriptor for the serial port
 * @return File descriptor; -1 in case of error
 */
int llOpenTransmitter(int fd);

```

```

/***
 * Function that opens and establishes the connection between the receiver and the transmitter
 * @param port Port name
 * @param role Flag that indicates the transmitter or the receiver
 * @return File descriptor; -1 in case of error
 */
int llopen(char* port, int role);

/***
 * Function that writes the information contained in the buffer to the serial port
 * @param fd File descriptor of the serial port
 * @param buffer Information to be written
 * @param length Length of the buffer
 * @return Number of characters written; -1 in case of error
 */
int llwrite(int fd, unsigned char* buffer, int length);

/***
 * Function that reads the information written in the serial port
 * @param fd File descriptor of the serial port
 * @param buffer Array of characters where the read information will be stored
 * @return Number of characters read; -1 in case of error
 */
int llread(int fd, unsigned char* buffer);

/***
 * Closes the connection for the receiver
 * @param fd File descriptor for the serial port
 * @return Positive value when sucess; negative value when error
 */
int llCloseReceiver(int fd);

/***
 * Closes the connection for the transmitter
 * @param fd File descriptor for the serial port
 * @return Positive value when sucess; negative value when error
 */
int llCloseTransmitter(int fd);

/***
 * Function that closes the connection between the receiver and the transmitter
 * @param fd File descriptor of the port
 * @param role Flag that indicates the transmitter or the receiver
 * @return Positive value when sucess; negative value when error
 */
int llclose(int fd, int role);

```

## files.c:

```
#include <stdio.h>
#include <stdlib.h>
#include "files.h"
#include "macros.h"

/** 
 * Auxiliary function to obtain the size of a file, from its file pointer
 * @param fp File pointer to the file
 * @return size of the file in question
 */
int getFileSize(FILE *fp){

    int lsize;

    fseek(fp, 0, SEEK_END);
    lsize = (int)fseek(fp);
    rewind(fp);

    return lsize;
}

/** 
 * Function that opens a file, from its name
 * @param fileName Name of the file to be opened
 * @param mode Mode in which to open the file
 * @return file pointer to the file in question
 */
FILE* openFile(char* fileName, char* mode){

    FILE* fp;
    fp = fopen(fileName, mode);

    if (fp == NULL)
    {
        perror(fileName);
        return NULL;
    }

    return fp;
}

/** 
 * Function that closes a file, from its file pointer
 * @param fp File pointer to the file to be closed
 * @return 0 if successful, EOF if an error occurs
 */
int closeFile(FILE* fp){
    return fclose(fp);
}
```

## files.h:

```
#pragma once

#include <stdio.h>

/** 
 * Auxiliary function to obtain the size of a file, from its file pointer
 * @param fp File pointer to the file
 * @return size of the file in question
 */
int getFileSize(FILE *fp);

/** 
 * Function that opens a file, from its name
 * @param fileName Name of the file to be opened
 * @param mode Mode in which to open the file
 * @return file pointer to the file in question
 */
FILE* openFile(char* fileName, char* mode);

/** 
 * Function that closes a file, from its file pointer
 * @param fp File pointer to the file to be closed
 * @return 0 if successful, EOF if an error occurs
 */
int closeFile(FILE* fp);
```

## macros.h:

```
#ifndef _MACROS_H_
#define _MACROS_H_

// ---- macros for data link layer ----
#define MAX_DATA_SIZE 1024 // max size of a data packet
#define MAX_PACK_SIZE (MAX_DATA_SIZE + 4) // max size of a data packet + 4 bytes for packet head
#define MAX_SIZE (MAX_PACK_SIZE + 6) // max size of data in a frame, + 4 bytes for packet head, + 6 bytes for frame header and tail
#define MAX_SIZE_FRAME (((MAX_PACK_SIZE + 1) * 2) + 5) // max size of a frame, with byte stuffing considered ((1029 * 2) + 5)
// 1029 -> all bytes that can suffer byte stuffing (and therefore be "duplicated"), that is, the packet and t
// 5 -> the bytes that won't (for sure) suffer byte stuffing (flags, bcc1, address and control bytes)

#define BUF_SIZE_SUP 5 // size of a supervision frame

#define BAUDRATE B38400 //38400 is the normal value
#define MODEMDEVICE "/dev/ttyS1"
#define _POSIX_SOURCE 1 /* POSIX compliant source */
#define FALSE 0
#define TRUE 1

#define TRANSMITTER 0
#define RECEIVER 1

#define SUPERVISION 0
#define INFORMATION 1

#define NUM_RETR 3
#define TIMEOUT 3

#define FLAG 0x7E
#define END_SEND 0x03
#define END_REC 0x01
#define S_0 0x00
#define S_1 0x40
#define SET 0x03
#define DISC 0x0B
#define UA 0x07
#define RR_0 0X05
#define RR_1 0X85
#define REJ_0 0x01
#define REJ_1 0x81
#define VTIME_VALUE 0
#define VMIN_VALUE 1

#define BYTE_STUFFING_ESCAPE 0x5D
#define BYTE_STUFFING_FLAG 0x5E
#define ESCAPE_BYT 0x7D
```

```

#define DATA_START    4

// ---- macros for application layer ----

#define CTRL_DATA      0x01
#define CTRL_START     0x02
#define CTRL_END       0x03

#define TYPE_FILESIZE   0x00
#define TYPE_FILENAME    0x01

#endif

```

## noncanonical.c:

```

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>
#include <unistd.h>
#include "macros.h"
#include "aux.h"
#include "data_link.h"
#include "app.h"

int main(int argc, char** argv)
{
    if( (argc < 2) ||
        ((strcmp("/dev/ttyS0", argv[1])!=0) &&
         (strcmp("/dev/ttyS1", argv[1])!=0) &&
         (strcmp("/dev/ttyS2", argv[1])!=0) &&
         (strcmp("/dev/ttyS3", argv[1])!=0) &&
         (strcmp("/dev/ttyS4", argv[1])!=0) ) {
        printf("Usage:\tnserial SerialPort\n\tex: nserial /dev/ttyS1\n");
        exit(1);
    }

    if(receiveFile(argv[1])<0){
        return -1;
    }

    return 0;
}

```

## statemachine.c:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>
#include <unistd.h>
#include "aux.h"
#include "statemachine.h"
#include "macros.h"

/***
 * Function to check if a byte is contained in the state machine's wantedBytes field
 * @param byte Byte to be checked
 * @param sm State machine for which to check
 * @return Index of the array where the byte was found; negative if byte is not a member
 */
int isWanted(unsigned char byte, state_machine_st* sm) {
    for (int i = 0; i < sm->wantedBytesLength; i++) {
        if (sm->wantedBytes[i] == byte)
            return i;
    }
    return -1;
}

/***
 * Function to update the state of the state machine
 * @param sm State machine to be updated
 * @param st State to be assigned to the state machine
 */
void change_state(state_machine_st* sm, state_st st) {
    sm->state = st;
}

/***
 * Function to create a state machine, with the given attributes
 * @param wantedBytes Possible bytes that are expected in the frame to be read by the state machine
 * @param wantedBytesLength Number of possible bytes that are expected in the frame to be read by the state machine
 * @param addressByte Address from which the frame to be read by the state machine is expected
 * @return Pointer to the new state machine "object" (struct)
 */
state_machine_st* create_state_machine(unsigned char* wantedBytes, int wantedBytesLength, unsigned char addressByte) {
    state_machine_st* sm = malloc(sizeof(state_machine_st));
    change_state(sm, START);
    sm->wantedBytes = wantedBytes;
    sm->wantedBytesLength = wantedBytesLength;
    sm->addressByte = addressByte;
    sm->dataLength = 0;
    return sm;
}

/***
 * Function to update the state machine according to the bytes read
 * @param sm State machine to be updated
 * @param byte Last byte to have been read, of the current frame
 * @param frame Address where the frame that's being read is being stored
 * @param mode Type of frame that's being read (Supervision or Information)
 */
void event_handler(state_machine_st* sm, unsigned char byte, unsigned char* frame, int mode) {
    static int i = 0;

    if(mode == SUPERVISION){
        switch(sm->state) {
            case START:
                if (byte == FLAG) {
                    change_state(sm, FLAG_RCV);
                    frame[0] = byte;
                }
                break;

            case FLAG_RCV:
                if (byte == FLAG)
                    break;
                else if (byte == sm->addressByte) {
                    change_state(sm, A_RCV);
                    frame[1] = byte;
                }
                else
                    change_state(sm, START);
                break;
        }
    }
}
```

```

        case A_RCV:
            if (byte == FLAG)
                | change_state(sm, FLAG_RCV);
            else {
                int n;
                if ((n = isWanted(byte, sm))>=0){
                    change_state(sm, C_RCV);
                    sm->foundIndex = n;
                    frame[2] = byte;
                }
                else
                    change_state(sm, START);
            }
            break;

        case C_RCV:
            if (byte == createBCC(frame[1], frame[2])){
                change_state(sm, BCC_OK);
                frame[3] = byte;
            }

            else if (byte == FLAG)
                | change_state(sm, FLAG_RCV);
            else
                | change_state(sm, START);
            break;

        case BCC_OK:
            if (byte == FLAG){
                change_state(sm, STOP);
                frame[4] = byte;
            }
            else
                | change_state(sm, START);
            break;

        default:
            | break;
    }
}

else if(mode == INFORMATION){

switch(sm->state) {

    case START:
        i = 0;
        if (byte == FLAG) {
            change_state(sm, FLAG_RCV);
            frame[i++] = byte;
        }
        break;

    case FLAG_RCV:
        if (byte == FLAG)
            break;
        else if (byte == sm->addressByte) {
            change_state(sm, A_RCV);
            frame[i++] = byte;
        }
        else {
            change_state(sm, START);
            i = (int) sm->state;
        }
        break;

    case A_RCV:
        if (byte == FLAG) {
            change_state(sm, FLAG_RCV);
            i = (int) sm->state;
        }
        else {
            if (isWanted(byte, sm) >= 0){
                change_state(sm, C_RCV);
                frame[i++] = byte;
            }
            else {
                change_state(sm, START);
                i = (int) sm->state;
            }
        }
        break;
}
}

```

```

        case C_RCV:
            if (byte == createBCC(frame[1], frame[2])){
                change_state(sm, BCC_OK);
                frame[i++] = byte;
            }
            else {
                if (byte == FLAG)
                    change_state(sm, FLAG_RCV);
                else
                    change_state(sm, START);

                i = (int) sm->state;
            }
            break;

        case BCC_OK:
            if(byte == FLAG){
                frame[i] = byte;
                change_state(sm, STOP);
                sm->dataLength = i-4;
            }
            else{
                frame[i++] = byte;
            }
            break;

        default:
            break;
    }

}

/***
 * Function to free the memory allocated to a state machine object
 * @param sm State machine to be destroyed
 */
void destroy_st(state_machine_st* sm) {
    free(sm);
}

```

## statemachine.h:

```

#pragma once

typedef enum state {
    START = 0,
    FLAG_RCV,
    A_RCV,
    C_RCV,
    BCC_OK,
    STOP
} state_st;

typedef struct state_machine {
    state_st state;
    unsigned char* wantedBytes;
    int wantedBytesLength;
    unsigned char addressByte;
    int foundIndex;
    int dataLength;
} state_machine_st;

/***
 * Function to check if a byte is contained in the state machine's wantedBytes field
 * @param byte Byte to be checked
 * @param sm State machine for which to check
 * @return Index of the array where the byte was found; negative if byte is not a member
 */
int isWanted(unsigned char byte, state_machine_st* sm);

/***
 * Function to update the state of the state machine
 * @param sm State machine to be updated
 * @param st State to be assigned to the state machine
 */
void change_state(state_machine_st* sm, state_st st);

/***
 * Function to create a state machine, with the given attributes
 * @param wantedBytes Possible bytes that are expected in the frame to be read by the state machine
 * @param wantedBytesLength Number of possible bytes that are expected in the frame to be read by the state machine
 * @param addressByte Address from which the frame to be read by the state machine is expected
 * @return Pointer to the new state machine "object" (struct)
 */
state_machine_st* create_state_machine(unsigned char* wantedBytes, int wantedBytesLength, unsigned char addressByte);

```

```

/** 
 * Function to update the state machine according to the bytes read
 * @param sm State machine to be updated
 * @param byte Last byte to have been read, of the current frame
 * @param frame Address where the frame that's being read is being stored
 * @param mode Type of frame that's being read (Supervision or Information)
 */
void event_handler(state_machine_st* sm, unsigned char byte, unsigned char* frame, int mode);

/** 
 * Function to free the memory allocated to a state machine object
 * @param sm State machine to be destroyed
 */
void destroy_st(state_machine_st* sm);

```

## writenoncanonical.c:

```

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>
#include <unistd.h>
#include "macros.h"
#include "aux.h"
#include "data_link.h"
#include "app.h"
#include "files.h"

int main(int argc, char** argv)
{
    if( (argc < 3) ||
        ((strcmp("/dev/ttyS0", argv[1])!=0) &&
        (strcmp("/dev/ttyS1", argv[1])!=0) &&
        (strcmp("/dev/ttyS2", argv[1])!=0) &&
        (strcmp("/dev/ttyS3", argv[1])!=0) &&
        (strcmp("/dev/ttyS4", argv[1])!=0 ))) {
        printf("Usage:\nserial SerialPort\n\tex: nserial /dev/ttyS1\n");
        exit(1);
    }

    if(sendFile(argv[1], argv[2])<0){
        return -1;
    }

    return 0;
}

```

## Anexo - Tabelas das medições da eficiência

FER	0	2	4	6	8	10
	0,77189	0,699982	0,5689	0,492899	0,38889	0,332244
	0,770041	0,667417	0,535191	0,377473	0,315441	0,264531
	0,769856	0,610329	0,521171	0,458877	0,374671	0,223677
	0,771364	0,65428	0,500317	0,490415	0,304857	0,377368
	0,769439	0,715227	0,523876	0,407921	0,378837	0,258878
	0,774322	0,61418	0,497759	0,431286	0,324724	0,390674
avg(S)	0,77	0,66	0,52	0,45	0,35	0,31

C	4800	9600	19200	38400
	0,780635	0,780537	0,780343	0,78001
	0,780636	0,780548	0,780356	0,780027
	0,780632	0,780052	0,780343	0,780014
	0,780629	0,780539	0,780345	0,78001
	0,780635	0,780531	0,780351	0,779993
	0,780636	0,780542	0,780347	0,779975
avg(S)	0,78063	0,78054	0,78035	0,78000

Packet size	32	64	128	256	512	1024	2048
	0,516905	0,618418	0,692849	0,739689	0,766405	0,780043	0,786205
	0,517178	0,618406	0,692761	0,739672	0,76639	0,780036	0,786219
	0,517424	0,61841	0,692863	0,739656	0,76638	0,780015	0,786198
	0,517147	0,618286	0,692851	0,739718	0,766412	0,780007	0,786213
	0,517169	0,618379	0,692915	0,739724	0,7664	0,780013	0,786207
	0,51591	0,61837	0,692758	0,739663	0,766392	0,780029	0,786226
avg(S)	0,52	0,62	0,69	0,74	0,77	0,78	0,79

T_prop	0	50	100	150	200	250
	0,780825	0,656738	0,567794	0,500024	0,446721	0,403691
	0,780878	0,656768	0,567769	0,500027	0,446717	0,403689
	0,780880	0,656737	0,567766	0,500028	0,446723	0,403683
	0,780884	0,656771	0,567773	0,500040	0,446731	0,403685
	0,780890	0,656742	0,567771	0,500053	0,446733	0,403687
	0,779037	0,656746	0,567789	0,500021	0,446728	0,403695
avg(S)	0,780867	0,656749	0,567776	0,500030	0,446726	0,403688