U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Redes de Computadores

1º Trabalho Laboratorial

Mestrado Integrado em Engenharia Informática e Computação

(15 de novembro de 2020)

João Rosário **up201806334**@fe.up.pt

João Castro Pinto **up201806667**@fe.up.pt

João Dossena **up201800174**@fe.up.pt

Índice

1 - Sumário

Este projeto foi realizado no âmbito da unidade curricular de Redes de Computadores do mestrado integrado de engenharia informática e computação. O objetivo do trabalho era implementar um protocolo de ligação de dados.

O nosso trabalho cumpre com todos os requisitos enunciados no guião fornecido e implementa um protocolo de ligação de dados fiável.

2 - Introdução

Para este projeto fomos desafiados a construir um protocolo de ligação de dados que permite a transmissão de dados entre dois computadores através de uma porta de série. Com o protocolo construído, devemos enviar implementar uma aplicação que utiliza o protocolo para enviar um ficheiro de um computador para outro.

Inicialmente, começamos por desenvolver o projeto através de uma porta de série virtual e após termos uma versão funcional do programa, passamos a trabalhar com os computadores das salas I320 e I321 da FEUP.

Segue-se uma descrição sucinta das secções seguintes.

- **Arquitetura -** Descrição das camadas funcionais e das interfaces entre elas.
- **Estrutura do código -** Explicação das APIs implementadas, das principais estruturas de dados utilizadas, das principais funções e sua relação com a arquitetura do programa.
- **Casos de uso principais -** Identificação dos mesmos e demonstração das sequências de chamada de funções.
- **Protocolo de ligação lógica -** Descrição dos principais aspetos funcionais e da estratégia de implementação dos mesmos com apresentação de extratos de código.
- **Protocolo de aplicação -** Descrição dos principais aspetos funcionais e da estratégia de implementação dos mesmos com apresentação de extratos de código.
- **Validação -** Apresentação dos testes efetuados com apresentação quantificada dos resultados.
- **Eficiência do protocolo de ligação de dados -** Estatísticas da eficiência do protocolo desenvolvido.
- **Conclusão -** Síntese da informação apresentada nas secções anteriores e reflexão sobre os objetivos de aprendizagem alcançados.

3 - Arquitetura

O programa foi implementado em duas camadas: a camada da aplicação e a camada da ligação lógica ou ligação de dados. Embora o desenvolvimento do projeto apenas passou pela a implementação destas duas camadas, foi utilizada uma

camada física como ponto de partida, que a sua interface foi pré-implementada pelo sistema operativo utilizado *Linux*.

O protocolo de transferência de dados está implementado pela camada da ligação lógica, enquanto que a camada da aplicação utiliza a camada da ligação lógica para enviar um ficheiro de um computador para o outro (ver anexo).

Um dos objetivos deste trabalho era garantir que existisse independência entre as diferentes camadas que compõem o protocolo. Durante o desenvolvimento do programa, tivemos esse pormenor em conta e criamos módulos diferentes e independentes entre si para a camada da aplicação e para a camada da ligação lógica.

A comunicação entre a camada da aplicação e a camada da ligação lógica é feita através de uma *API* descrita numa das secções seguintes. Já entre as camadas menos abstratas, recorreu-se à *API* fornecida pelo *Linux*.

4 - Estrutura de código

Para organizar o código e garantir a independência entre camadas criamos módulos diferentes para as camadas implementadas por nós, o módulo representado pela pasta *dataLayer* e o módulo representado pela pasta *appLayer*. Utilizamos também um módulo de auxílio a que chamamos *utils*.

A organização dos módulos e dos ficheiros de código segue a representação da figura 2.
Tal como é possível observar, são utilizados no total 10 ficheiros de código.

4.1 - Módulo "AppLayer"

O módulo *AppLayer* é a implementação da camada da aplicação. Este módulo tem acesso à parte pública de *dataLayer*. Para iniciar a aplicação, e com isso, iniciar também as camadas inferiores, chama-se a função `appRun()` na função `main()` após o tratamento dos parâmetros do programa.

A primeira função que `appRun()` chama é `llopen()` do módulo *DataLayer* que assegura a boa ligação entre o emissor e o recetor.

Posteriormente consoante a aplicação esteja num estado de emissor ou receptor é chamada a função `sendFile()` ou a função `receiveFile()`, respetivamente. Estas funções tratam de enviar o ficheiro indicado como parâmetro do programa ou receber o ficheiro enviado pelo emissor.

Após o envio do ficheiro ou da sua receção, a função `appRun()` assegura o fecho do protocolo de ligação da porta de série, chamando a função `llclose`.

Para fazer medições de eficiência recorremos a `getTimeofDay()` que foi chamada antes e depois da receção ou envio do ficheiro.

4.2 - Módulo "DataLayer"

Este módulo é constituído principalmente pelas funções `llopen`, `llread`, `llwrite`, `llclose`. As funções deste módulo usam funções de "nível" mais baixo

nomeadamente o write e read, pelo que decidimos criar um *dataLayerPrivate* que tem apenas as funções chamadas pelas 4 funções principais para garantir o encapsulamento deste módulo.

Para evidenciar bem a independência de camadas no nosso projeto, dividimos módulo *dataLayer* em duas partes: as funções públicas e as funções privadas. Esta ideia de encapsulamento é útil para garantir que as únicas funções acessíveis por fora de *dataLayer* sejam as funções `llopen`, `llwrite`, `llread` e `llclose`. *DataLayer* não depende de *appLayer* e o módulo da aplicação apenas tem acesso às funções públicas.

4.3 - Módulo "utils"

O módulo *utils* tem 5 funções que foram necessárias em vários locais do código, pelo que para não haver uma repetição destas mesmas, elas foram incluídas neste pequeno módulo de forma a preservar a integridade e legibilidade do código.

As funções aqui presentes são as que se seguem: `ceiling` (retorna o valor arredondado em excesso), `bit()`, `getBit()`, `printString()` e `displayStats()`.

5 - Casos de uso principais

Os casos de uso principais deste projeto são o envio e a receção de um ficheiro através de uma porta de série. Segue-se uma pequena descrição das funções utilizadas para a implementação destes dois casos de uso:

5.1 - Emissor/Recetor

5.1.1 - Funções principais da camada de ligação

`llopen` - Estabelece o protocolo de ligação
`llwrite` - Envia tramas de informação
`llread` - Recebe tramas de informação
`llclose` - Termina o protocolo de ligação

5.1.2 - Funções principais da camada de aplicação

`sendFile` - Envia o ficheiro pretendido em formato de pacotes com a ajuda da função llwrite

`createControlPacket` - Cria um pacote de controlo do ficheiro segundo as indicações do guião

`createDataPacket` - Cria um pacote de dados do ficheiro segundo as indicações do guião

`receiveFile` - Recebe pacotes de informação e os guarda em um ficheiro

`parseControlPacket` - Recebe um pacote de controlo e verifica se está correto

`parseDataPacket` - Recebe um pacote de dados e verifica se está correto

5.2 - Interface

Durante o envio, vão sendo imprimidas no ecrã, tanto do lado do transmissor como do lado do recetor, as ações relevantes à transmissão do ficheiro, para que o utilizador consiga perceber o que está a acontecer no programa. Esta *interface* com o utilizador consideramos outro caso de uso relevante.

Ao executar o programa, temos a possibilidade de colocar diversos parâmetros. Consideramos que fosse necessário da parte do utilizador indicar se o programa deve correr em modo recetor ou modo transmissor e ainda qual é a porta de série que deve utilizar. No caso em que o programa corra em modo transmissor, o utilizador ainda deve indicar qual o ficheiro que quer transmitir. Para além destes parâmetros obrigatórios, também fornecemos a possibilidade ao utilizador de parametrizar o *baud rate* utilizado na transferência de dados, o tamanho máximo da trama e o número de segundos que o emissor demora a fazer *timeout* (explicado na secção 6.2.2).

6 - Protocolo de ligação lógica

O protocolo da ligação lógica e as comunicações desta camada foram desenvolvidas com base no protocolo *Stop & Wait* estudado nas aulas teóricas.

6.1 - Descrição detalhada das funções públicas

Sempre que nas seguintes descrições se referir o envio ou receção de uma trama de informação, são utilizadas para esse efeito as funções `sendIFrame` e `receiveIMessage`. Para o envio ou receção de tramas não numeradas ou de supervisão são utilizadas as funções `sendNotIFrame` e `receiveNotIMessage`.

6.1.1 - `int llopen(char *port, int appStatus)`

A função `llopen` estabelece a ligação entre o emissor e o recetor. Ver anexo II.7. No nosso caso, a função altera o seu funcionamento para o recetor e o emissor. Esta informação é recebida pela função no parâmetro *appStatus*. Tanto no emissor como no recetor, inicialmente o `llopen` começa por abrir um descritor de ficheiro com o caminho *port* para a porta de série.

Quando `llopen` é corrida pelo emissor, envia a trama de controlo SET construída pela função `buildSETFrame` e espera uma resposta do recetor. Quando recebida, o emissor termina a função.

Por sua vez, o recetor recebe a trama SET e envia uma resposta: a trama de controlo UA construída pela função `buildUAFrame`. Enviando a resposta, o recetor termina o `llopen`.

A função `llopen` retorna o descritor de ficheiro aberto que representa a porta de série e em caso de erro retorna um valor negativo.

6.1.2 - `int llwrite(int fd, char * buffer, int length)`

A função `llwrite` envia um conjunto de *length* bytes indicados pelo parâmetro *buffer* para o descritor *fd* de forma fiável com recurso à utilização de tramas. Ver anexo II.8.

Inicialmente, `llwrite` transforma a informação a enviar numa trama de informação com o auxílio da função `prepareIFrame`. O mecanismo de *byte stuffing* para garantir a transparência das tramas é implementado na função `stuffFrame`, que é chamada pela função `prepareIFrame`. De seguida, `llwrite` chama a função `sendIFrame` que se responsabiliza de enviar a trama.

A função `llwrite` retorna -1 se a escrita não se realizar corretamente e retorna 0 se tiver enviado a trama de informação e recebido a resposta correta.

6.1.3 - `int llread(int fd, char * buffer)`

Esta função é responsável pela receção das tramas de informação provenientes do descritor *fd* e por colocar a informação recebida na trama no *array* recebido como parâmetro, *buffer*. Ver anexo II.8.

Inicialmente, `llread` começa por receber uma trama de informação pelo descritor *fd* com a ajuda da função receiveIMessage.

Após a receção da trama de informação, é enviada uma resposta ao emissor sobre o estado em que a trama foi recebida. Na situação em que a trama tenha sido recebida sem erros, é enviada a trama RR de resposta ao transmissor.

Para situações em que a trama de informação chegue ao recetor com erros, `llread` envia uma resposta do tipo REJ para pedir o reenvio da trama ao emissor.

Tal como é pedido no guião, implementamos os casos de receção de tramas duplicadas, caso haja ou não hajam erros.

`llread` retorna 0 se tiver efetuado a leitura da trama com sucesso e -1 se tiver ocorrido um erro.

6.1.4 - `int llclose(int fd, char * buffer)`

A função `llclose` trata de fechar a ligação entre emissor e recetor. Ver anexo II.9. A função, tal como no `llopen`, altera o seu funcionamento para o recetor e o emissor. Tanto no emissor como no recetor, inicialmente o `llclose` começa por abrir um descritor de ficheiro com o caminho *port* para a porta de série.

Quando `llclose` é corrida pelo emissor, envia a trama de controlo DISC construída pela função `buildDISCFrame` e espera uma resposta do recetor. Quando recebida, o emissor envia uma trama de controlo UA construída pela função `buildUAFrame` e termina a função.

Por sua vez, o recetor recebe a trama DISC e envia uma resposta: a trama de controlo DISC construída pela função `buildUAFrame`. Enviando a resposta, o recetor espera a trama de resposta UA e termina o `llclose`.

A função `llclose` retorna 1 em caso de sucesso e um valor negativo correspondente ao erro em caso de insucesso.

6.2 - Descrição detalhada das funções privadas

Os valores de retorno destas funções estão especificados em comentários no código.

6.2.1 - `int receiveIMessage(frame_t *frame, int fd)`

Este função implementa uma máquina de estados para receber uma trama pela descritor de ficheiro *fd*. Cada *byte* da trama é lido e processado individualmente, tendo em conta o estado da receção e o *byte* recebido. Para perceber melhor a forma como a máquina de estados foi implementada, o leitor pode consultar os anexos.

6.2.2 - `int sendIFrame(frame_t *frame, int fd)`

Dentro da função `sendIFrame`, o envio da trama é seguido de uma receção de uma resposta (ver anexo II.8) enviada pelo recetor da trama. Tanto o envio da trama de informação como a receção da resposta respetiva, estão dentro de um ciclo que apenas se repete quando a resposta recebida não é uma trama do tipo RR ou quando o tempo que a resposta demora a ser recebida excede o número de segundos de *timeout* passado como parâmetro do programa (ver secção 5.3). O ciclo termina quando é recebida uma resposta RR com a identificação correta da trama ou quando são excedidas o número de tentativas de escrita de trama. No final desta função, chamamos a função `destuffFrame` para fazer *byte unstuffing* à trama recebida.

6.2.3 - `int receiveNotIMessage(frame_t *frame, int fd, int responseId, int timeout)`

O funcionamento desta função é muito semelhante ao de `receiveIMessage`, com a exceção de que não precisamos de tratar a receção do campo de dados e do seu campo de controlo. Também utiliza uma máquina de estados.

O parâmetro *responseId* é utilizado para quando esta função é chamada para receção de uma resposta e deve-se indicar no parâmetro qual a *id* que a resposta deve ter. Quando não se quer receber uma resposta, passa-se a macro *RESPONSE_WITHOUT_ID*.

O parâmetro timeout tem um funcionamento semelhante: quando se quer que haja timeouts na leitura de uma trama não numerada ou de supervisão, coloca-se neste parâmetro o número de segundo que o programa pode demorar a ler e processar um byte antes que ocorra um timeout. Se não for necessário ter esta funcionalidade ativada, passa-se a macro *NO_TIMEOUT*.

6.2.4 - `int sendNotIFrame(frame_t *frame, int fd)`

A implementação desta função é muito simples: apenas escreve a trama recebida como parâmetro no descritor *fd.*

7 - Protocolo de aplicação

Sempre que nas seguintes descrições se referir o envio ou receção de uma pacote, as funções que são utilizadas para esse efeito são `llwrite` e `llread`, respetivamente.

7.1 - Descrição detalhada das funções

7.1.1 - `int sendFile(char * filename)`

Esta função trata de abrir o ficheiro com nome *filename*, dividi-lo em pacotes e enviar cada uma pela porta de série com recurso à função `llwrite`. Antes e depois de enviar o ficheiro, envia também um pacote de controlo para sinalizar o início ou fim do ficheiro, respetivamente, e o nome e tamanho do ficheiro.

7.1.2 - `int receiveFile()`

A função `receiveFile` está encarregada de ler da porta de série o ficheiro que o emissor enviou, separado em pacotes, e guardá-lo em disco. Antes e depois da receção de todos os pacotes do ficheiro são recebidos os pacotes de controlo que o emissor envia na função `sendFile`.

A leitura dos pacotes do ficheiro está dentro de um ciclo for que itera um número de vezes correspondente ao número de tramas expectável.

7.1.3 - `packet_t createDataPacket(u_int8_t *string, int number, int size)`

A função `createDataPacket` recebe como argumentos um *array* de *bytes*, uma numeração e um tamanho de pacote. Com esses parâmetros, cria um pacote como é pedido no enunciado: *flag* de pacote de dados, numeração, 2 octetos para o tamanho, e a seguir todos os dados. Por fim, retorna o pacote preenchido.

7.1.4 - `int parseDataPacket(u_int8_t *packetArray, u_int8_t *bytes)`

A função `parseDataPacket` recebe como argumentos um pacote e um *array* de *bytes*. Primeiro lê o tamanho do pacote, e depois faz parse dos seus dados e retorna-os por referência no *array* de octetos. Por fim, retorna o tamanho do pacote.

7.1.5 - `packet_t createControlPacket(u_int8_t type, unsigned size, char *filename)`

A função `createControlPacket` recebe como argumentos uma *flag* que especifica o tipo de pacote, um tamanho e um nome de ficheiro. Com esses parâmetros, cria um pacote como é pedido no enunciado em formato TLV: a *flag* de pacote de controlo (*START* ou *END*), *flag* de tamanho, número de *bytes* do tamanho, octetos para o tamanho, *flag* de nome do ficheiro, tamanho do nome do ficheiro e nome do ficheiro. Por fim, retorna o pacote preenchido.

7.1.6 - `int parseControlPacket(u_int8_t *controlPacket, unsigned* fileSize, char *filename)`

A função `parseControlPacket` recebe como argumentos um pacote, seu tamanho e um array de caracteres. Verifica a flag do pacote, lê o tamanho e lê o nome do ficheiro. Por fim, retorna por referência o tamanho e o nome do ficheiro e retorna a flag de controlo por valor.

7.1.7 - `void appRun()`

Esta função inicia a aplicação chamando por ordem as funções `llopen`, `sendFile` ou `receiveFile` dependendo se estiver a correr o emissor ou recetor do programa e `llclose`. Só há uma chamada a `appRun()` em todo o código e está na função `main()`.

8 - Validação

A aplicação foi posta à prova com os seguintes testes :

- Envio e receção corretas de ficheiros variados;
- Interrupção temporária da ligação durante o envio do ficheiro;
- Variação do tamanho máximo das tramas;
- Variação do *baud rate*;
- Gerou-se curto circuito enquanto se envia o ficheiro para testar ruído;
- Erros induzidos.

No nosso caso a aplicação superou os testes todos.

9 - Eficiência do protocolo de ligação de dados

Tal como nos foi proposto no guião deste trabalho, avaliamos a eficiência do nosso programa em vários parâmetros. Todas as medições estão registadas na folha de cálculo do anexo II.6.

9.1 - Variação de "Frame Error Ratio"

Colocando as medições efetuadas num gráfico da eficiência (S = R/C) em função de FER, percebe-se que há uma diminuição de eficiência com o aumento do Frame Error Ratio.

9.2 - Variação de "Baudrate"

Como o Baudrate (C) é a taxa de símbolos por tempo, e o débito (R) é a taxa de bits por tempo, os dois acabam por variar de maneira semelhante. Portanto a eficiência não tem alterações significativas (S = R/C).

9.3 - Variação do tamanho máximo da trama

Analisando o gráfico de eficiência em função do tamanho máximo da trama, fica claro que quando se aumenta o tamanho máximo da trama, a eficiência aumenta.

10 - Conclusões

O trabalho pretendia que fosse criada uma aplicação que possibilitasse a transferência de ficheiros. Podemos afirmar que o objetivo foi concluído com sucesso visto que como referido no ponto 8 todos os objetivos foram alcançados.

Este projeto fez-nos compreender como um protocolo de transmissão de dados fiável pode funcionar. Os três alunos ficaram entendidos sobre a matéria.

Anexos I - Código fonte

main.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 <unistd.h>
#include <time.h>
#include <signal.h>
#include "appLayer/applicationLayer.h"
#include "dataLayer/dataLayer.h"
#include "macros.h"
#include "utils/utils.h"

application app;
int baudrate = BAUDRATE;
extern int timeoutLength;
unsigned maxFrameSize = MAX_FRAME_SIZE;
unsigned maxFrameDataLength = MAX_FRAME_DATA_LENGTH;
unsigned maxPacketLength = MAX_PACKET_LENGTH;
unsigned maxPacketDataLength = MAX_PACKET_DATA_LENGTH;

int main(int argc, char *argv[])
{
    srand(time(NULL));

    system("umask 0077");

    if (argc == 3 || argc == 6) {
        if ((strcmp("-r", argv[1]) != 0)) {
            printf("Receiver usage: %s -r <port> [baudrate
maxFrameSize timeout]\nTransmitter usage: %s -s <port> <filename>
[baudrate maxFrameSize timeout]\n", argv[0], argv[0]);
            printf("Valid baudrates are:\n0, 50, 75, 110, 134, 150,
200, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600,
115200\n");
            exit(1);
        }
    }
    else if (argc == 4 || argc == 7) {
        if ((strcmp("-s", argv[1]) != 0)) {
```

```
            printf("Receiver usage: %s -r <port> [baudrate
maxFrameSize timeout]\nTransmitter usage: %s -s <port> <filename>
[baudrate maxFrameSize timeout]\n", argv[0], argv[0]);
            printf("Valid baudrates are:\n0, 50, 75, 110, 134, 150,
200, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600,
115200\n");
            exit(1);
        }
    }
    else {
        printf("Receiver usage: %s -r <port> [baudrate maxFrameSize
timeout]\nTransmitter usage: %s -s <port> <filename> [baudrate
maxFrameSize timeout]\n", argv[0], argv[0]);
        printf("Valid baudrates are:\n0, 50, 75, 110, 134, 150, 200,
300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600,
115200\n");
        exit(1);
    }

    if (strcmp("-s", argv[1])== 0) {
        app.status = TRANSMITTER;
        if(argc == 7) {
            baudrate = convertBaudrate(atoi(argv[4]));
            maxFrameSize = atoi(argv[5]);
            if (maxFrameSize > 128000) {
                printf("MAIN - Max value for frame size is 128000.
Setting it to default 512...\n");
                maxFrameSize = 512;
            }
            timeoutLength = atoi(argv[6]);
            maxFrameDataLength = (maxFrameSize - 8);
            maxPacketLength = maxFrameDataLength;
            maxPacketDataLength = maxFrameDataLength - 4;
        }
    }
    else if (strcmp("-r", argv[1])== 0) {
        app.status = RECEIVER;
        if(argc == 6) {
            baudrate = convertBaudrate(atoi(argv[3]));
            maxFrameSize = atoi(argv[4]);
            if (maxFrameSize > 128000) {
                printf("MAIN - Max value for frame size is 128000.
Setting it to default 512...\n");
                maxFrameSize = 512;
            }
            timeoutLength = atoi(argv[5]);
```

```
            maxFrameDataLength = (maxFrameSize - 8);
            maxPacketLength = maxFrameDataLength;
            maxPacketDataLength = maxFrameDataLength - 4;
        }
    }

    if (app.status == TRANSMITTER) {
        strcpy(app.port, argv[2]);
        strcpy(app.filename, argv[3]);
    }
    else if (app.status == RECEIVER) {
        strcpy(app.port, argv[2]);
    }

    printf("MAIN - Starting app...\n");

    appRun();

    printf("MAIN - Closing app...\n");

    return 0;
}
```

applicationLayer.h

```
typedef struct {
    int fd; // serial port file descriptor
    int status; // transmitter or receiver
    char port[20];  // port name
    char filename[MAX_FILENAME_LENGTH];  // name of file to send
} application;


typedef struct {
    u_int8_t **bytes;   // we had memory problems, and got it
working this way: we used it an array of bytes
    int size;
} packet_t;
```

dataLayer.h

```
#pragma once
#include "../macros.h"
```

```
typedef struct {
    u_int8_t **bytes;   // we had memory problems, and got it
working this way: we used it an array of bytes
    int size;
    int infoId;         // frame id, if needed
} frame_t;


int llopen(char *port, int appStatus);
int llclose(int fd);
int llread(int fd, char *buffer);
int llwrite(int fd, char * buffer, int length);

int clearSerialPort(char *port);

int convertBaudrate(int baudArg);
```

dataLayerPrivate.h

```
#pragma once
#include "../utils/utils.h"

typedef enum {
    INIT,
    RCV_FLAG,
    RCV_A,
    RCV_C,
    RCV_BCC1,
    RCV_DATA,
    RCV_BCC2,
    COMPLETE
} receive_state_t; // auxiliary to receive functions



// ---

int receiveIMessage(frame_t *frame, int fd);
int receiveNotIMessage(frame_t *frame, int fd, int responseId, int
timeout);
int sendIFrame(frame_t *frame, int fd);
int sendNotIFrame(frame_t *frame, int fd);


u_int8_t bccCalculator(u_int8_t bytes[], int start, int length);
```

```
bool bccVerifier(u_int8_t bytes[], int start, int length, u_int8_t
parity);

void buildSETFrame(frame_t *frame, bool transmitterToReceiver);
bool isSETFrame(frame_t *frame);
void buildUAFrame(frame_t * frame, bool transmitterToReceiver);
bool isUAFrame(frame_t *frame);
void buildDISCFrame(frame_t * frame, bool transmitterToReceiver);
bool isDISCFrame(frame_t *frame);
void stuffFrame(frame_t * frame);
void destuffFrame(frame_t *frame);

void prepareI(frame_t *info, char* data, int size);
void prepareResponse(frame_t *frame, bool valid, int id);

void prepareFrameDataSize(int frameSize, u_int8_t *sizeBytes);
void prepareToReceive(frame_t *frame, int size);
void printFrame(frame_t *frame);

void readTimeoutHandler(int signo);
```

applicationLayer.c

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.#include <sys/types.h>
#include <sys/time.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 "applicationLayer.h"
#include "../dataLayer/dataLayer.h"
#include "../utils/utils.h"
```

```
extern application app;
extern int maxFrameDataLength;
extern int maxPacketLength;
extern int maxPacketDataLength;

void appRun() {
    if ((app.fd = llopen(app.port, app.status)) < 0) {
        printf("DATA - Error in llopen: %d\n", app.fd);
        // clearSerialPort(app.port);
        exit(1);
    }
    struct timeval begin, end;
    gettimeofday(&begin, NULL);
    switch (app.status) {
        case TRANSMITTER:;
            sendFile(app.filename);
        break;
        case RECEIVER:;
            receiveFile();
        break;
    }
    gettimeofday(&end, NULL);

    int llcloseReturn = llclose(app.fd);
    if (llcloseReturn < 0) {
        printf("DATA - Error in llclose: %d\n", llcloseReturn);
        // clearSerialPort(app.port);
        exit(1);
    }

    displayStats(begin, end);
}

int sendFile(char * filename){
    printf("APP - Starting to send file...\n");
    packet_t packet;

    packet.bytes = (u_int8_t **) malloc(sizeof(u_int8_t *));
    (*(packet.bytes)) = (u_int8_t *)malloc(maxPacketLength);

    int fileFd;

    fileFd = open(filename, O_RDONLY | O_NONBLOCK);
    if (fileFd == -1) {
        perror("DATA - file not opened correctly");
        return -1;
```

```
    }

    struct stat st;
    stat(filename, &st);
    unsigned fileSize = st.st_size;

    packet = createControlPacket(START, fileSize, filename);

    if(llwrite(app.fd, (char *)(*(packet.bytes)), packet.size) < 0){
        printf("APP - Error transmitting start control packet in
applicationLayer.c ...\n");
        return -1;
    }

    int size = 0, number = 0;
    u_int8_t *buffer = (u_int8_t *)malloc(maxPacketLength);
    do {
        size = read(fileFd, buffer, maxPacketLength/2 - 4);

        if(size == 0) break;

        printf("APP - Read a data packet from file.\n");
        packet = createDataPacket(buffer, (number % 256), size);

        if(llwrite(app.fd, (char *)(*(packet.bytes)), packet.size) <
0){
            printf("APP - Error transmitting data packet in
applicationLayer.c ...\n");
            return -1;
        }
        number++;
    } while (size > 0);
    packet = createControlPacket(END, fileSize, filename);
    printf("APP - Sent whole file.\n");

    if(llwrite(app.fd, (char *)(*(packet.bytes)), packet.size) < 0){
        printf("APP - Error transmitting end control packet in
applicationLayer.c ...\n");
        return -1;
    }

    close(fileFd);
    return 0;
}
```

```
packet_t createControlPacket(u_int8_t type, unsigned size, char *
filename){
    packet_t packet;
    packet.bytes = (u_int8_t **) malloc(sizeof(u_int8_t *));
    (*(packet.bytes)) = (u_int8_t *)malloc(maxPacketLength);
    (*(packet.bytes)) = (u_int8_t *)malloc(maxPacketLength);
    (*(packet.bytes))[0] = type;
    (*(packet.bytes))[1] = FILESIZE;
    (*(packet.bytes))[2] = 4;
    (*(packet.bytes))[3] = (u_int8_t)(size >> 24);
    (*(packet.bytes))[4] = (u_int8_t)(size >> 16);
    (*(packet.bytes))[5] = (u_int8_t)(size >> 8);
    (*(packet.bytes))[6] = (u_int8_t)size; //LSB
    (*(packet.bytes))[7] = FILENAME;
    (*(packet.bytes))[8] = strlen(filename) + 1;         // got to
have +1
    for(int i = 0; i < (*(packet.bytes))[8] ; i++){
        (*(packet.bytes))[9 + i] = filename[i];
    }
    packet.size = 8 + (*(packet.bytes))[8];

    return packet;
}

int parseControlPacket(u_int8_t* controlPacket, unsigned* fileSize,
char* filename){
    int controlStatus = controlPacket[0];
    if(controlStatus != START && controlStatus != END){
        printf("APP - Unknown control packet status: %d - not START
nor END!\n" , controlStatus);
        return -1;
    }

    *fileSize = controlPacket[3] << 24 | controlPacket[4] << 16 |
controlPacket[5] << 8 | controlPacket[6];
    u_int8_t stringSize = controlPacket[8];
    for(int i = 0; i < stringSize; i++){
        filename[i] = controlPacket[8 + 1 + i];
    }

    return controlStatus;
}

packet_t createDataPacket(u_int8_t * string, int number, int size){
    packet_t packet;
    packet.bytes = (u_int8_t **) malloc(sizeof(u_int8_t *));
```

```
    (*(packet.bytes)) = (u_int8_t *)malloc(maxPacketLength/2);
    packet.size = size + 4;
    (*(packet.bytes))[0] = DATA;
    (*(packet.bytes))[1] = number;
    (*(packet.bytes))[2] = (int) (size / 256);
    (*(packet.bytes))[3] = size % 256;
    for(int i = 0; i < size ; i++){
        (*(packet.bytes))[4 + i] = string[i];
    }

    return packet;
}

int parseDataPacket(u_int8_t * packetArray, u_int8_t * bytes) {
    int packetDataSize = packetArray[2]*256 + packetArray[3];
    for (int i = 0; i < packetDataSize; i++) {
        bytes[i] = packetArray[i + 4];
    }
    return packetDataSize;
}

int receiveFile(){
    printf("APP - Starting to receive file...\n");

    char *receive = (char *)malloc(maxPacketLength);
    if(llread(app.fd, receive) < 0){
        printf("APP - Error receiving start control packet in
applicationLayer.c ...\n");
        return -1;
    }

    unsigned fileSize;
    int controlStatus;
    char filename[MAX_FILENAME_LENGTH];
    controlStatus = parseControlPacket((u_int8_t *)receive,
&fileSize, filename);

    if(controlStatus != START){
        printf("APP - Error receiving start control packet in
applicationLayer.c ...\n");
        return -1;
    }

    if(controlStatus == START){
        printf("APP - Received START Control Packet ...\n");
    }
```

```
    int fileFd = open(filename, O_WRONLY | O_CREAT , S_IRWXG |
S_IRWXU | S_IRWXO);
    if (fileFd <= -1) {
        perror("file not opened correctly");
        return -1;
    }


    u_int8_t *bytes = (u_int8_t *)malloc(maxPacketLength/2 - 4);
    for(int i = 0 ; i < (fileSize / (maxPacketLength/2 - 4)) + 1;
i++){

        if(llread(app.fd, receive) < 0){
            printf("APP - Error receiving data packet in
applicationLayer.c ...\n");
            return -1;
        }
        int packetDataSize = parseDataPacket((u_int8_t *)receive,
bytes);
        if(write(fileFd, bytes, packetDataSize) < 0){
            perror("APP - Error writing to file ...");
            return -1;
        }
        printf("APP - Wrote a data packet to file.\n");
    }

    printf("APP - Received whole file.\n");


    if(llread(app.fd, receive) < 0){
        printf("APP - Error receiving end control packet in
applicationLayer.c ...\n");
        return -1;
    }

    controlStatus = parseControlPacket((u_int8_t *)receive,
&fileSize, filename);

    if(controlStatus != END){
        printf("APP - Error receiving end control packet in
applicationLayer.c ...\n");
        return -1;
    }
```

```
    if(controlStatus == END){
        printf("APP - Received END Control Packet ...\n");
    }

    close(fileFd);
    return 0;

}

void printPacket(packet_t *packet) {
    printf("\nStarting printPacket...\n\tSize: %d\n", packet->size);
    for (int i = 0; i < packet->size; i++)
    {
        printf("\tByte %d: %x\n", i, (*(packet->bytes))[i]);
    }
    printf("printPacket ended\n\n");
}
```

dataLayer.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 <unistd.h>
#include <signal.h>
#include <time.h>
#include "dataLayer.h"
#include "dataLayerPrivate.h"

int status;
int timeoutLength;
extern int idFrameSent;
extern int lastFrameReceivedId;
extern int baudrate;

extern int maxFrameSize;
extern int maxFrameDataLength;


int llopen(char *port, int appStatus)
```

```
{
    printf("DATA - Entered llopen\n");
    status = appStatus;

    struct termios oldtio, newtio;

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

    if (tcgetattr(fd, &oldtio) == -1) {
        perror("tcgetattr");
        return -2;
    }


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

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

    newtio.c_cc[VTIME] = 0; // time to time-out in deciseconds
    newtio.c_cc[VMIN] = 1;  // min number of chars to read

    if (tcsetattr(fd, TCSANOW, &newtio) == -1) {
        perror("tcsetattr");
        return -3;
    }

    frame_t setFrame;
    frame_t responseFrame;
    frame_t receiverFrame;
    frame_t uaFrame;
    setFrame.bytes = (u_int8_t **)malloc(sizeof(u_int8_t *));
    responseFrame.bytes = (u_int8_t **)malloc(sizeof(u_int8_t *));
    receiverFrame.bytes = (u_int8_t **)malloc(sizeof(u_int8_t *));
    uaFrame.bytes = (u_int8_t **)malloc(sizeof(u_int8_t *));

    (*(setFrame.bytes)) = (u_int8_t *)malloc(maxFrameSize);
    (*(responseFrame.bytes)) = (u_int8_t *)malloc(maxFrameSize);
    (*(receiverFrame.bytes)) = (u_int8_t *)malloc(maxFrameSize);
```

```
    (*(uaFrame.bytes)) = (u_int8_t *)malloc(maxFrameSize);


    switch (appStatus) {
        case TRANSMITTER:;
            for (int i = 0;; i++) {
                if (i == MAX_FRAME_RETRANSMISSIONS) {
                    printf("DATA - Max Number of retransmissions
reached. Exiting program.\n");
                    return -1;
                }

                buildSETFrame(&setFrame, true);

                if (sendNotIFrame(&setFrame, fd)) {
                    perror("sendNotIFrame\n");
                    return -5;
                }
                prepareToReceive(&responseFrame, 5);
                int responseReceive =
receiveNotIMessage(&responseFrame, fd, RESPONSE_WITHOUT_ID,
timeoutLength);


                if (responseReceive == -1) continue;         // in a
timeout, retransmit frame
                else if (responseReceive < -2) {printf("Error in
receiveNotIMessage from llopen\n");  return -7;}
                if (!isUAFrame(&responseFrame)) continue;       //
wrong frame received

                break;
            }
            break;
        case RECEIVER:;
            prepareToReceive(&receiverFrame, 5);
            int error = receiveNotIMessage(&receiverFrame, fd,
RESPONSE_WITHOUT_ID, NO_TIMEOUT);
            if (error) {
                printf("DATA - ReceiveNotIMessage returned %d\n",
error);
                return -7;
            }
            if (!isSETFrame(&receiverFrame)) {
                printf("DATA - Frame is not of type SET\n");
                return -8;
```

```
            }

            buildUAFrame(&uaFrame, true);
            if (sendNotIFrame(&uaFrame, fd)) {
                printf("DATA - Problem in sendNotIFrame\n");
                return -5;
            }
            break;
    }
    printf("DATA - Opened serial port connection\n");

    return fd;
}

int llclose(int fd) {
    printf("DATA - Entered llclose\n");
    frame_t discFrame;
    frame_t receiveFrame;
    frame_t uaFrame;

    discFrame.bytes = (u_int8_t **)malloc(sizeof(u_int8_t *));
    receiveFrame.bytes = (u_int8_t **)malloc(sizeof(u_int8_t *));
    uaFrame.bytes = (u_int8_t **)malloc(sizeof(u_int8_t *));

    (*(discFrame.bytes)) = (u_int8_t *)malloc(maxFrameSize);
    (*(receiveFrame.bytes)) = (u_int8_t *)malloc(maxFrameSize);
    (*(uaFrame.bytes)) = (u_int8_t *)malloc(maxFrameSize);


    int receiveReturn;
    switch (status) {
        case TRANSMITTER:;
            for (int i = 0;; i++) {
                if (i == MAX_FRAME_RETRANSMISSIONS) {
                    printf("DATA - Max Number of retransmissions
reached. Exiting program.\n");
                    return -1;
                }

                buildDISCFrame(&discFrame, true);
                if (sendNotIFrame(&discFrame, fd)) return -2;

                prepareToReceive(&receiveFrame, 5);
                receiveReturn = receiveNotIMessage(&receiveFrame,
fd, RESPONSE_WITHOUT_ID, timeoutLength);
                if (receiveReturn == -1) continue;        //in a
```

```
timeout, retransmit frame
                else if (receiveReturn < -1) return -4;
                if (!isDISCFrame(&receiveFrame)) continue;      //
wrong frame received

                buildUAFrame(&uaFrame, true);
                if (sendNotIFrame(&uaFrame, fd)) return -2;

                break;
            }
        break;
        case RECEIVER:;
            for (int i = 0; i < MAX_READ_ATTEMPTS; i++) {
                prepareToReceive(&receiveFrame, 5);
                int receiveReturn =
receiveNotIMessage(&receiveFrame, fd, RESPONSE_WITHOUT_ID,
timeoutLength);
                if (receiveReturn == -1) continue;
                else if (receiveReturn) return -7;
                if (!isDISCFrame(&receiveFrame)) return -5;
                break;
            }
            buildDISCFrame(&discFrame, true);
            if (sendNotIFrame(&discFrame, fd)) return -2;
            prepareToReceive(&receiveFrame, 5);
            int receiveNotIMessageReturn =
receiveNotIMessage(&receiveFrame, fd, RESPONSE_WITHOUT_ID,
NO_TIMEOUT);
            if (receiveNotIMessageReturn) return -4;
            if (!isUAFrame(&receiveFrame)) return -5;
        break;
    }
    if (close(fd) == -1) return -8;
    printf("DATA - Closed serial port connection\n");
    return 1;
}


int llread(int fd, char * buffer){
    frame_t frame, response;
    frame.bytes = (u_int8_t **) malloc(sizeof(u_int8_t *));
    response.bytes = (u_int8_t **) malloc(sizeof(u_int8_t *));
    (*(frame.bytes)) = (u_int8_t *)malloc(maxFrameSize);
    (*(response.bytes)) = (u_int8_t *)malloc(maxFrameSize);
    int receiveIMessageReturn, sameReadAttempts = 1;
```

```
    do {
        receiveIMessageReturn = receiveIMessage(&frame, fd);
        if (receiveIMessageReturn < -3 || receiveIMessageReturn > 1)
{
            printf("DATA - receiveIMessage returned unexpected
value\n");
            return -1;
        }
        if (receiveIMessageReturn >= 0) {
            prepareResponse(&response, true, (frame.infoId + 1) %
2);
            printf("DATA - Sent RR frame to the transmitter\n");
            sameReadAttempts = 0;
        }
        else if (receiveIMessageReturn == -1 ||
receiveIMessageReturn == -2) {
            if (lastFrameReceivedId != -1 && frame.infoId ==
lastFrameReceivedId) {
                prepareResponse(&response, true, (frame.infoId + 1)
% 2);
                printf("DATA - Read a duplicate frame\nDATA - Sent
RR frame to the transmitter\n");
                sameReadAttempts = 0;
            }
            else {
                prepareResponse(&response, false, (frame.infoId + 1)
% 2);
                printf("DATA - Sent REJ frame to the
transmitter\n");
                sameReadAttempts++;
            }
        }
        if (receiveIMessageReturn >= 0) {

            lastFrameReceivedId = frame.infoId;
        }
        if (receiveIMessageReturn != -3) {
            if (sendNotIFrame(&response, fd) == -1) return -1;
        }
        if (receiveIMessageReturn == 0){
            memcpy(buffer, (*(frame.bytes)) + 6, (*(frame.bytes))[4]
* 256 + (*(frame.bytes))[5]);

        }
```

```
        if (receiveIMessageReturn == -3) {
            printf("DATA - Serial Port couldn't be read. Exiting
llread...\n");
            return -1;
        }
    } while (receiveIMessageReturn != 0 && sameReadAttempts <
MAX_READ_ATTEMPTS);

    // if (sameReadAttempts == MAX_READ_ATTEMPTS) {
    //     printf("DATA - Max read attempts of the same frame
reached.\n");
    //     return -1;
    // }
    return 0;
}

int llwrite(int fd, char * buffer, int length)
{

    frame_t info;
    prepareI(&info, buffer, length); //Prepara a trama de informação

    if (sendIFrame(&info, fd) == -1) return -1;
    return 0;
}


int clearSerialPort(char *port) {
    printf("Clearing serial port in case of errors. Quit program
with Ctrl-C\n");

    int auxFd = open(port, O_RDWR | O_NOCTTY);
    if (auxFd == -1) {
        perror("DATA - error clearing serialPort");
        return 1;
    }

    struct termios oldtio, newtio;

    if (tcgetattr(auxFd, &oldtio) == -1) {
        perror("tcgetattr");
        return -2;
    }


    bzero(&newtio, sizeof(newtio));
```

```
    newtio.c_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;
    newtio.c_iflag = IGNPAR;
    newtio.c_oflag = 0;

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

    newtio.c_cc[VTIME] = 0; // time to time-out in deciseconds
    newtio.c_cc[VMIN] = 1;  // min number of chars to read

    if (tcsetattr(auxFd, TCSANOW, &newtio) == -1) {
        perror("tcsetattr");
        return -3;
    }

    char c;
    while (read(auxFd, &c, 1) != 0) printf("DATA - byte cleared:
%x\n", c);
    if (close(auxFd) == -1) return 2;
    return 0;
}


int convertBaudrate(int baudArg) {
    switch(baudArg) {
        case 0:         return 0000000;         /* hang up */
        case 50:        return 0000001;
        case 75:        return 0000002;
        case 110:       return 0000003;
        case 134:       return 0000004;
        case 150:       return 0000005;
        case 200:       return 0000006;
        case 300:       return 0000007;
        case 600:       return 0000010;
        case 1200:      return 0000011;
        case 1800:      return 0000012;
        case 2400:      return 0000013;
        case 4800:      return 0000014;
        case 9600:      return 0000015;
        case 19200:     return 0000016;
        case 38400:     return 0000017;
        case 57600:     return 0010001;
        case 115200:    return 0010002;
        default:
            printf("\nArgument is not a valid baudrate. Using
default Baudrate 38400\n");
```

```
            printf("Valid baudrates are:\n0, 50, 75, 110, 134, 150,
200, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600,
115200\n\n");
            break;
    }
    return 0000017;
}
```

dataLayerPrivate.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 <unistd.h>
#include <time.h>
#include <signal.h>
#include <errno.h>
#include "dataLayer.h"
#include "dataLayerPrivate.h"
#include "../macros.h"
#include "../utils/utils.h"

int idFrameSent = 0;
int lastFrameReceivedId = -1;
extern int timeoutLength;
extern int maxFrameSize;
extern int maxFrameDataLength;

void stuffFrame(frame_t * frame)
{
    int stuffingCounter = 0;
    u_int8_t *frameRealData = (u_int8_t
*)malloc(maxFrameDataLength);
    for (int i = 0; i < 6; i++) frameRealData[i] =
(*(frame->bytes))[i];
    for (int i = 6; i < frame->size - 2 + stuffingCounter; i++) {
        if ((*(frame->bytes))[i - stuffingCounter] == FLAG) {
            frameRealData[i] = ESC;
            frameRealData[++i] = FLAG_STUFFING;
            stuffingCounter++;
            continue;
```

```
        }
        if ((*(frame->bytes))[i - stuffingCounter] == ESC) {
            frameRealData[i] = ESC;
            frameRealData[++i] = ESC_STUFFING;
            stuffingCounter++;
            continue;
        }
        frameRealData[i] = (*(frame->bytes))[i - stuffingCounter];
    }
    for (int i = frame->size - 2; i < frame->size; i++)
frameRealData[i + stuffingCounter] = (*(frame->bytes))[i];


    frame->size += stuffingCounter;

    frameRealData[4] = (frame->size - 8) / 256;
    frameRealData[5] = (frame->size - 8) % 256;

    memcpy((*(frame->bytes)), frameRealData, frame->size);
}


void destuffFrame(frame_t *frame) {
    bool destuffing = false;
    int destuffingCounter = 0;
    u_int8_t *frameRealData = (u_int8_t
*)malloc(maxFrameDataLength);
    for (int i = 0; i < 6; i++) frameRealData[i] =
(*(frame->bytes))[i];
    for (int i = 6; i < frame->size - 2; i++) {
        if ((*(frame->bytes))[i] == ESC) {
            destuffingCounter++;
            destuffing = true;
            continue;
        }
        if (destuffing && (*(frame->bytes))[i] == FLAG_STUFFING) {
            frameRealData[i - destuffingCounter] = FLAG;
            destuffing = false;
            continue;
        }
        else if (destuffing && (*(frame->bytes))[i] == ESC_STUFFING)
{
            frameRealData[i - destuffingCounter] = ESC;
            destuffing = false;
            continue;
        }
```

```
        frameRealData[i - destuffingCounter] = (*(frame->bytes))[i];
    }
    for (int i = frame->size - 2; i < frame->size; i++)
frameRealData[i - destuffingCounter] = (*(frame->bytes))[i];

    frame->size -= destuffingCounter;

    frameRealData[4] = (frame->size - 8) / 256;
    frameRealData[5] = (frame->size - 8) % 256;

    frameRealData[frame->size - 2] = bccCalculator(frameRealData, 4,
frameRealData[4] * 256 + frameRealData[5] + 2);

    memcpy((*(frame->bytes)), frameRealData, frame->size);
}


// pode ser necessário ter os dados em mais que uma frame
void prepareI(frame_t *info, char* data, int length) //Testar
{
    u_int8_t frameDataSize[2];
    info->bytes = (u_int8_t **) malloc(sizeof(u_int8_t *));
    (*(info->bytes)) = (u_int8_t *)malloc(maxFrameSize);

        //debug
    (*(info->bytes))[0] = FLAG; //F
    (*(info->bytes))[1] = TRANSMITTER_TO_RECEIVER; //A
    (*(info->bytes))[2] = idFrameSent << 6 | I;
    info->infoId = idFrameSent;
    (*(info->bytes))[3] = bccCalculator((*(info->bytes)), 1, 2);
//BCC1, calculado com A e C


    prepareFrameDataSize(length, frameDataSize);
    (*(info->bytes))[4] = frameDataSize[0];
    (*(info->bytes))[5] = frameDataSize[1];

    for (int j = 0; j < length; j++) {
        (*(info->bytes))[6 + j] = data[j];
    }

    int bcc2_byte_ix = 4 + 2 + ((*(info->bytes))[4] << 8)  +
(*(info->bytes))[5];

    (*(info->bytes))[bcc2_byte_ix] = bccCalculator((*(info->bytes)),
4, ((*(info->bytes))[4] << 8) + (*(info->bytes))[5] + 2);
```

```
    (*(info->bytes))[bcc2_byte_ix + 1] = FLAG;
    info->size = 4 + 2 + (*(info->bytes))[4] * 256 +
(*(info->bytes))[5] + 2;

    stuffFrame(info);

    idFrameSent = (idFrameSent + 1) % 2;

}



// Returns -3 if there is an error with reading from the serial port
// Returns -2 if there is an error with bcc2
// Returns -1 if there is an error with data size value
// Returns 0 if received ok
// Returns 1 if received a repeated frame
int receiveIMessage(frame_t *frame, int fd){
    u_int8_t c;
    receive_state_t state = INIT;
    int dataCounter = -2, returnValue = 0, bcc2Size = 1;
    do {

        int bytesRead = read(fd, &c, 1);

        if (bytesRead < 0) {
            perror("read error");
            return -3;
        }
        switch (state) {
            case INIT:
                if (c == FLAG) {
                    state = RCV_FLAG;
                    (*(frame->bytes))[0] = c;
                }
                break;
            case RCV_FLAG:
                if (c == TRANSMITTER_TO_RECEIVER || c ==
RECEIVER_TO_TRANSMITTER) {
                    state = RCV_A;
                    (*(frame->bytes))[1] = c;
                }
                else if (c == FLAG) {
                    state = RCV_FLAG;
                }
                else {
```

```
                    state = INIT;
                }
                break;
            case RCV_A:
                if ((c & I_MASK) == I) {
                    state = RCV_C;
                    (*(frame->bytes))[2] = c;
                    frame->infoId = c >> 6;
                }
                else if (c == FLAG)
                    state = RCV_FLAG;
                else
                    state = INIT;
                break;
            case RCV_C:
                if (bccVerifier((*(frame->bytes)), 1, 2, c)) {
                    state = RCV_BCC1;
                    (*(frame->bytes))[3] = c;
                }
                else if (c == FLAG)
                    state = RCV_FLAG;
                else {
                    state = INIT;
                }
                break;
            case RCV_BCC1:
                (*(frame->bytes))[4 + 2 + dataCounter] = c;
                dataCounter++;
                if (dataCounter == ((*(frame->bytes))[4] * 256 +
(*(frame->bytes))[5])) state = RCV_DATA;
                break;
            case RCV_DATA:
                if (bcc2Size == 1) {
                    state = RCV_BCC2;
                    (*(frame->bytes))[4 + 2 + dataCounter] = c;
                }
                else if (bcc2Size == 2) {
                    state = RCV_BCC2;
                    (*(frame->bytes))[4 + 2 + dataCounter + 1] = c;
                }
                else if (c == FLAG)
                    state = RCV_FLAG;
                else {
                    printf("DATA - BCC2 not correct\n");
                    returnValue = -2;
                }
```

```
                if (c == FLAG && bcc2Size == 1) {
                    bcc2Size = 2;
                    state = RCV_DATA;
                }
                break;
            case RCV_BCC2:
                if (c == FLAG) {
                    state = COMPLETE;
                    (*(frame->bytes))[4 + 2 + dataCounter +
bcc2Size] = c;
                }
                else
                    state = INIT;
                break;
            case COMPLETE: break;
        }

    } while (state != COMPLETE && returnValue == 0);
    if (lastFrameReceivedId != -1 && lastFrameReceivedId ==
frame->infoId && returnValue == 0) {
        printf("DATA - Read a duplicate frame\n");
        returnValue = 1;
    }
    else if (returnValue == 0) {
        frame->size = 4 + 2 + dataCounter + bcc2Size + 1;
        printf("DATA - Received %d bytes\n", frame->size);
        destuffFrame(frame);
        returnValue = 0;
    }
    return returnValue;
}



void readTimeoutHandler(int signo) { return; }

// Returns 0 if received ok
// Returns 1 if received RR ok
// Returns 2 if received REJ ok
// Returns -1 if there was a timeout
// Returns -2 if there was a read error
int receiveNotIMessage(frame_t *frame, int fd, int responseId, int
timeout)
{
    u_int8_t c;
```

```
    receive_state_t state = INIT;

    struct sigaction sigAux;
    sigAux.sa_handler = readTimeoutHandler;
    sigaction(SIGALRM, &sigAux, NULL);
    siginterrupt(SIGALRM, 1);

    do {
        if (timeout != NO_TIMEOUT)
            alarm(timeout);

        int bytesRead = read(fd, &c, 1);
        if (bytesRead < 0) {
            if (errno == EINTR) {
                printf("DATA - Timeout occured while reading a
frame!\n");
                return -1;
            }
            perror("read error");
            return -2;
        }

        switch (state) {
            case INIT:
                if (c == FLAG) {
                    state = RCV_FLAG;
                    (*(frame->bytes))[0] = c;
                }
                break;
            case RCV_FLAG:
                if (c == TRANSMITTER_TO_RECEIVER || c ==
RECEIVER_TO_TRANSMITTER) {
                    state = RCV_A;
                    (*(frame->bytes))[1] = c;
                }
                else if (c == FLAG) {
                    state = RCV_FLAG;
                }
                else {
                    state = INIT;
                    return -3;
                }
                break;
            case RCV_A:
                if ((c == SET) || (c == UA) || (c == DISC) || (c ==
(RR | (responseId << 7))) || (c == (REJ | (responseId << 7)))) {
```

```
                    state = RCV_C;
                    (*(frame->bytes))[2] = c;
                }
                else if (c == FLAG) {
                    state = RCV_FLAG;
                }
                else
                    state = INIT;
                break;
            case RCV_C:
                if (bccVerifier((*(frame->bytes)), 1, 2, c)) {
                    state = RCV_BCC1;
                    (*(frame->bytes))[3] = c;
                }
                else if (c == FLAG)
                    state = RCV_FLAG;
                else {
                    printf("DATA - BCC1 not correct\n");
                    state = INIT;
                }
                break;
            case RCV_BCC1:
                if (c == FLAG) {
                    state = COMPLETE;
                    (*(frame->bytes))[4] = c;
                }
                else
                    state = INIT;
                break;
            case COMPLETE:
                break;
            default:
                printf("DATA - Unknown state");
                state = INIT;
                break;
        }
    } while (state != COMPLETE);

    alarm(0);   // cancel any pending alarm() calls

    frame->size = 5;
    int returnValue = 0;
    if ((*(frame->bytes))[2] == (RR | (responseId << 7)))
returnValue = 1;
    if ((*(frame->bytes))[2] == (REJ | (responseId << 7)))
returnValue = 2;
```

```
    return returnValue;
}

// Returns -1 if timeout, 0 if ok
int sendNotIFrame(frame_t *frame, int fd) {
    int writeReturn = write(fd, (*(frame->bytes)), frame->size);
    printf("DATA - %d bytes sent\n", writeReturn);

    if (writeReturn == -1) return -1;
    return 0;
}

// Returns -1 if max write attempts were reached
// Returns 0 if ok
int sendIFrame(frame_t *frame, int fd) {
    int attempts = 0, sentBytes = 0;
    frame_t responseFrame;
    responseFrame.bytes = (u_int8_t **)malloc(sizeof(u_int8_t *));
    (*(responseFrame.bytes)) = (u_int8_t *)malloc(maxFrameSize);
    while (1) {
        if(attempts >= MAX_WRITE_ATTEMPTS)
        {
            printf("DATA - Too many write attempts\n");
            return -1;
        }
        if ((sentBytes = write(fd, (*(frame->bytes)), frame->size))
== -1) return -1;
        printf("DATA - %d bytes sent\n", sentBytes);


        int receiveReturn = receiveNotIMessage(&responseFrame, fd,
(frame->infoId + 1) % 2, timeoutLength);

        if (receiveReturn == -1) {
            printf("DATA - Timeout reading response, trying
again...\n");
        }
        else if (receiveReturn == -2) {
            printf("DATA - There was a reading error while reading
response, trying again...\n");
        }
        else if (receiveReturn == 0) {
            printf("DATA - Received wrong response, trying
again...\n");
        }
```

```
        else if (receiveReturn == 1) {
            printf("DATA - Received an OK message from the
receiver.\n");
            break;
        }
        else if (receiveReturn == 2) {
            printf("DATA - Received not OK message from the
receiver, trying again...\n");
        }
        else {
            printf("DATA - Unexpected response frame, trying
again...\n");
        }
        attempts++;
    }
    return 0;
}

void prepareResponse(frame_t *frame, bool valid, int id) {
    frame->size = 5;
    (*(frame->bytes))[0] = FLAG;
    (*(frame->bytes))[1] = TRANSMITTER_TO_RECEIVER;
    if (valid)
        (*(frame->bytes))[2] = RR | (id << 7);
    else
        (*(frame->bytes))[2] = REJ | (id << 7);
    (*(frame->bytes))[3] = bccCalculator((*(frame->bytes)), 1, 2);
    (*(frame->bytes))[4] = FLAG;
}

// ---

u_int8_t bccCalculator(u_int8_t bytes[], int start, int length)
{
  int bcc = 0x00;
    for (int i = start; i < start + length; i++)
    {
        bcc ^= bytes[i];
    }
    return bcc;
}

// Return true if bcc verifies else otherwise
bool bccVerifier(u_int8_t bytes[], int start, int length, u_int8_t
parity)
{
```

```
    return (bccCalculator(bytes, start, length) == parity);
}

void buildSETFrame(frame_t *frame, bool transmitterToReceiver)
{
    frame->size = 5;
    (*(frame->bytes))[0] = FLAG;
    if (transmitterToReceiver)
        (*(frame->bytes))[1] = TRANSMITTER_TO_RECEIVER;
    else
        (*(frame->bytes))[1] = RECEIVER_TO_TRANSMITTER;
    (*(frame->bytes))[2] = SET;
    (*(frame->bytes))[3] = bccCalculator((*(frame->bytes)), 1, 2);
// BCC
    (*(frame->bytes))[4] = FLAG;
}

bool isSETFrame(frame_t *frame) {
    if (frame->size != 5) return false;
    return (*(frame->bytes))[2] == SET;
}

void buildUAFrame(frame_t *frame, bool transmitterToReceiver)
{
    frame->size = 5;
    (*(frame->bytes))[0] = FLAG;
    if (transmitterToReceiver)
        (*(frame->bytes))[1] = TRANSMITTER_TO_RECEIVER;
    else
        (*(frame->bytes))[1] = RECEIVER_TO_TRANSMITTER;
    (*(frame->bytes))[2] = UA;
    (*(frame->bytes))[3] = bccCalculator((*(frame->bytes)), 1, 2);
// BCC
    (*(frame->bytes))[4] = FLAG;
}

bool isUAFrame(frame_t *frame) {
    if (frame->size != 5) return false;
    return (*(frame->bytes))[2] == UA;
}

void buildDISCFrame(frame_t *frame, bool transmitterToReceiver)
{
    frame->size = 5;
    (*(frame->bytes))[0] = FLAG;
    if (transmitterToReceiver)
```

```
        (*(frame->bytes))[1] = TRANSMITTER_TO_RECEIVER;
    else
        (*(frame->bytes))[1] = RECEIVER_TO_TRANSMITTER;
    (*(frame->bytes))[2] = DISC;
    (*(frame->bytes))[3] = bccCalculator((*(frame->bytes)), 1, 2);
// BCC
    (*(frame->bytes))[4] = FLAG;
}

bool isDISCFrame(frame_t *frame) {
    if (frame->size != 5) return false;
    return (*(frame->bytes))[2] == DISC;
}

void prepareToReceive(frame_t *frame, int size)
{
    frame->size = size;
}

void prepareFrameDataSize(int frameSize, u_int8_t *sizeBytes) {
    sizeBytes[0] = (u_int8_t) (frameSize >> 8);
    sizeBytes[1] = (u_int8_t) frameSize;
}

void printFrame(frame_t *frame) {
    printf("\nStarting printFrame...\n\tSize: %d\n", frame->size);
    for (int i = 0; i < frame->size; i++)
    {
        printf("\tByte %d: %x \n", i, (*(frame->bytes))[i]);
    }
    printf("printFrame ended\n\n");
}
```

utils.h

```
#pragma once
#include <sys/time.h>

typedef enum {
    false,
    true
} bool;

int ceiling(float x);
```

```
u_int64_t bit(unsigned n);

u_int8_t getBit(int byte, int b);

void printString(char *str);

void displayStats(struct timeval begin, struct timeval end);
```

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

int ceiling(float x) {
    int y = x;
    if (y == x) return x;
    return x + 1;
}

u_int64_t bit(unsigned n) {
    return 1 << n;
}

u_int8_t getBit(int byte, int b)
{
    return (byte >> b) & bit(0);
}


void printString(char *str)
{
    printf("\nStarting printString...\n\tSize: %ld\n", strlen(str));
    for (int i = 0; i < strlen(str); i++)
    {
        printf("\tstr[%d]: %c\n", i, str[i]);
    }
    printf("printString ended\n");
```

```
}


void displayStats(struct timeval begin, struct timeval end) {
    printf("\n\nPROGRAM STATS:\nExecution in seconds: %lf\n",
(end.tv_sec - begin.tv_sec) + (end.tv_usec -
begin.tv_usec)/1000000.0);
}
```

macros.h

```
#pragma once

#define BAUDRATE B38400
#define _POSIX_SOURCE 1 /* POSIX compliant source */

#define FLAG 0x7e           // F

#define TRANSMITTER_TO_RECEIVER 0x03    // A
#define RECEIVER_TO_TRANSMITTER 0x01

#define SET 0x03        // C
#define DISC 0x0b
#define UA 0x07
#define RR 0x05  // 0b R 0 0 0 0 1 0 1
#define REJ 0x01 // 0b R 0 0 0 0 0 0 1
#define I 0x00   // 0b 0 S 0 0 0 0 0 0

#define DATA 1
#define START 2
#define END 3
#define FILESIZE 0
#define FILENAME 1
#define RR_MASK 0x7f
#define REJ_MASK 0x7f
#define I_MASK 0xbf

#define ESC 0x7d
#define FLAG_STUFFING 0x5e
#define ESC_STUFFING 0x5d
#define FLAG_MORE_FRAMES_TO_COME 0xaa
#define FLAG_LAST_FRAME 0xbb

#define BYTE_MASK 11111111
```

```
#define RECEIVER 0
#define TRANSMITTER 1

#define MAX_WRITE_ATTEMPTS 3
#define MAX_READ_ATTEMPTS 3

#define MAX_FRAME_SIZE 512                          // minimum is 16
#define MAX_FRAME_DATA_LENGTH (MAX_FRAME_SIZE - 8)     // if all
bytes are stuffed it takes the MAX_FRAME_SIZE
#define MAX_FRAME_RETRANSMISSIONS 5

#define MAX_PACKET_LENGTH (MAX_FRAME_DATA_LENGTH)
#define MAX_PACKET_DATA_LENGTH (MAX_FRAME_DATA_LENGTH - 4)

#define MAX_FILENAME_LENGTH 512

#define RESPONSE_WITHOUT_ID -1
#define NO_TIMEOUT -1
```

Anexos II

II.1

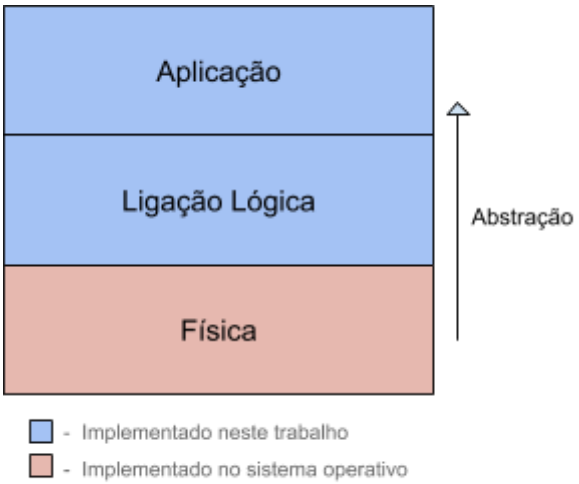
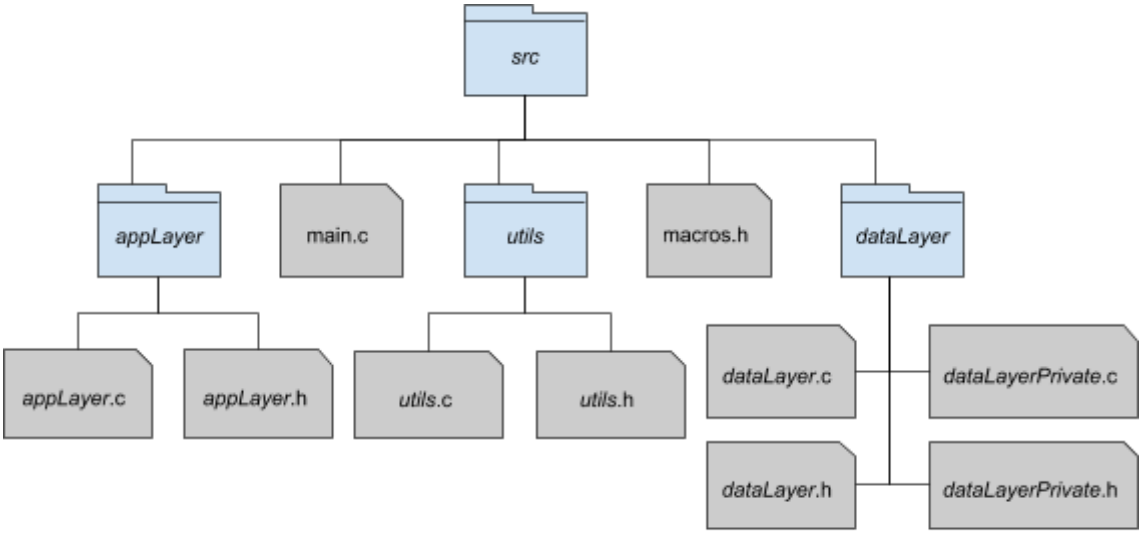


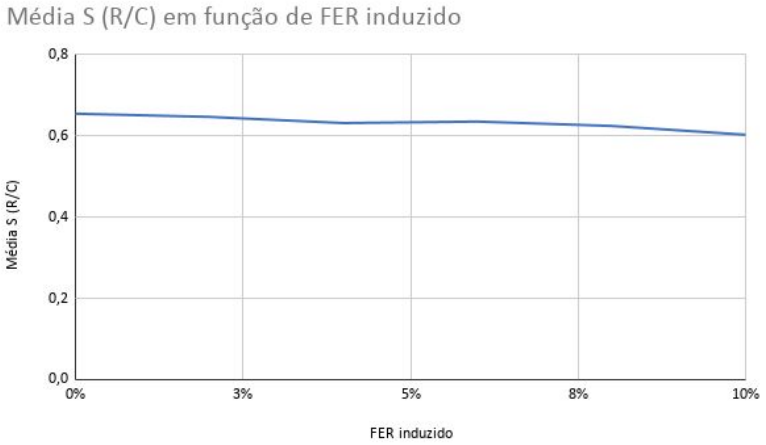
Figura 1 - Camadas do programa

II.2

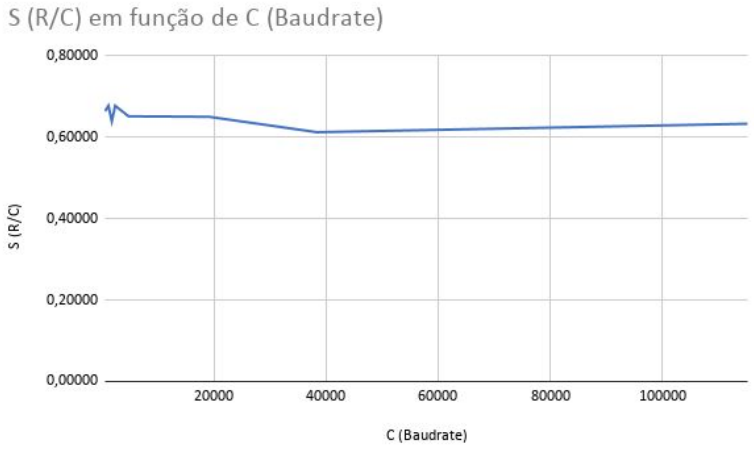


Anexo II.2 - Organização do código

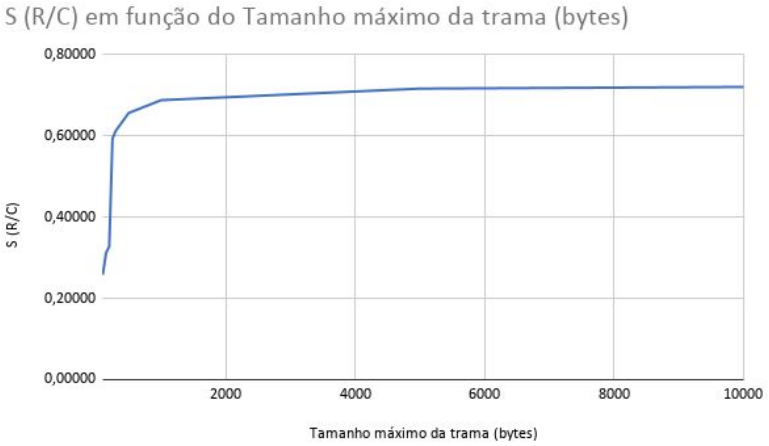
II.3


Média S (R/C) em função de FER induzido
0,8
0,6
0,4
0,2
0,0
Média S (R/C)
0%
3%
5%
8%
10%
FER induzido

II.4


S (R/C) em função de C (Baudrate)
0,80000
0,60000
0,40000
0,20000
0,00000
S (R/C)
20000
40000
60000
80000
100000
C (Baudrate)

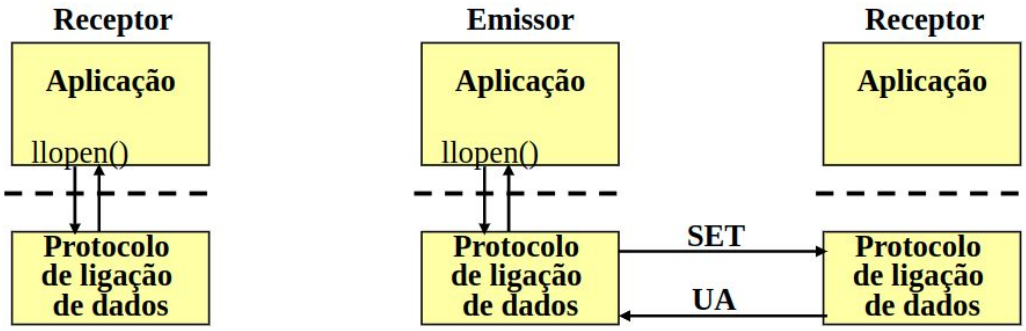
II.5


S (R/C) em função do Tamanho máximo da trama (bytes)
0,80000
0,60000
0,40000
0,20000
0,00000
S (R/C)
2000
4000
6000
8000
10000
Tamanho máximo da trama (bytes)

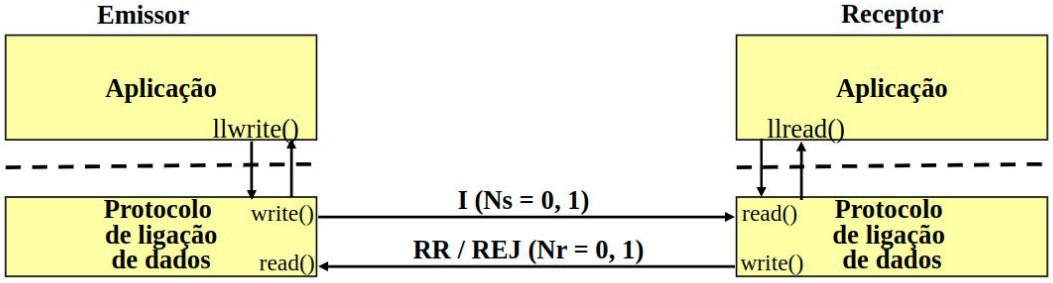
II.6

Link para o Google Sheet com os registos das medições do programa

II.7



II.8



II.9