U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Redes de Computadores 2021/2022

1º Trabalho laboratorial: Ligação de dados

Realizado por:

- Angela Manuela Correia Antelo Costa Cruz up201806781
- Fabio Huang up201806829

Sumário

O presente trabalho foi realizado no âmbito da disciplina de Redes de Computadores (RCOM) do 3ºano do Mestrado Integrado de Engenharia Informática e Computação (MIEIC) da Universidade do Porto.Trata-se do primeiro trabalho laboratorial da disciplina com objetivo de implementar de um protocolo de ligações de dados e simples de transferência de ficheiros entre duas máquinas.

O trabalho prático foi concluído com sucesso, tendo sido desenvolvida uma aplicação capaz de transmitir ficheiros entre dois computadores sem qualquer perda ou erro.

Introdução

O primeiro trabalho laboratorial tinha como objetivo a implementação de um protocolo de ligação de dados. Na prática, foi criada uma aplicação de transferência de ficheiros entre dois computadores, recorrendo a uma porta série.

Este relatório tem como objetivo a exposição e explicação das componentes teóricas utilizadas para a realização do projeto. O relatório seguirá a seguinte estrutura:

- **Arquitetura:** exibição dos blocos funcionais presentes
- **Estrutura do código**: demonstração das principais estruturas de dados, funções e a sua relação com a arquitetura
- **Casos de uso principais:** demonstração das sequências de chamada de funções
- **Protocolo de ligação lógica:** identificação dos principais aspetos funcionais e descrição da estratégia de implementação dos mesmos
- **Protocolo de aplicação:** identificação dos principais aspetos funcionais e descrição da estratégia de implementação dos mesmos
- **Validação:** Descrição dos testes efetuados
- **Eficiência do protocolo de ligação de dados:** caracterização estatística da eficiência do protocolo e elementos de valorização trazidos a aplicação
- **Conclusão**: síntese da informação apresentada e reflexões sobre os objetivos.

Arquitetura

O código do projeto está dividido em duas partes , a parte do protocol e da application recorrendo também a ficheiros auxiliares para recorrer a state machine e algumas mensagens de erro usadas no meio do programa.

A parte do protocolo está encarregue da comunicação entre as portas série e a aplicação do programa. Esta é a camada que trata da preparação dos dados e dos tratamentos dos possíveis erros como também a escrita , envio, e ligação dos dados.

A parte da aplicação é responsável por fazer a transferência dos ficheiros de acordo com os dados que foram introduzidos pelo utilizador no terminal.

Estrutura do Código

application.c |application.h

Encarregue de verificar os dados que foram inseridos pelo utilizador na linha de comandos, verificar se estão no formato devido para iniciar a transferência de dados. Tem a função ParseArgs() que verifica então a validade dos dados. A função sendPacket() e receivePacket() e também a função main().

protocol.c | protocol.h

Ficheiro que contém funções da camada de ligação de dados como llopen(), llwrite(), llread() e llclose() que são descritas mais abaixo na seção de protocolo de ligação de dados. É nesta parte do código que também temos as funções receiver_UA() que envia o sinal UA ao transmitter e a função transmitter_SET() que envia a data do emissor para o receptor.

stateMachine.c | stateMachine.h

Ficheiro com states machines usadas no ficheiro de protocol.c . Contem updateStateMachine_CONNECTION() para leitura das tramas nas funçoes llwrite(), llopen() **.** updateStateMachine_COMMUNICATION() usada para a leitura das tramas de informação na função llread(). E por último temos a função updateStateMachine_CLOSE() que é usada na função llclose().

utils.c |utils.h

Mensagens de erros foram necessárias para garantir que o programa fazia o controle de vários erros. Também contém a função de handler do timeout.

macros.h

Ficheiro com macros definidas que foram usadas várias vezes no programa.

Casos de uso principais

Para utilizar a nossa aplicação é necessário compilar os ficheiros com o comando `**make**`, depois para iniciar será necessário entrar na linha do comando `**./application /dev/ttyS<N>**`, onde o N e' o número da porta, por exemplo `**./application /dev/ttyS0**`, depois a aplicação irá perguntar a identidade (emissor ou receptor), aqui no lado do receptor, basta entrar na linha de comando o número `**1**` para correr a aplicação como receptor, neste caso de seguida irá pedir também para introduzir o nome do ficheiro pretendido para enviar ao emissor (por exemplo **pinguim.gif**), e no lado do emissor, introduzir na linha de comando o número `**2**`. É recomendado iniciar primeiro a aplicação de recetor antes que a aplicação do emissor. Depois irá aparecer na consola as várias informações sobre o estado atual da aplicação.

Protocolo de ligação lógica

O protocolo de ligação de dados implementado tem como objetivo configurar a porta série ,estabelecer a ligação e a terminação pela porta série e fazer a escrita e leitura de dados (fazendo o stuffing e destuffing dos dados) e controlar os erros neste processo.

As funções usadas para fazer este processo são:

llopen()- Função responsável em estabelecer a ligação entre o emissor e receptor mas também para configurar e modificar as definições da porta série com as funções tcgetattr() e tcsetattr(). Começando pelo emissor por enviar uma trama de controlo SET e aguardar pela resposta do receptor, durante a espera é ativado o temporizador, sendo este desativado quando receber a resposta UA. Caso não receba resposta UA dentro do tempo definido (TIME_OUT_SCS), SET é reenviado um número máximo de vezes (MAX_TRIES), caso

ultrapasse este número o programa do lado do emissor termina com erro de TIMEOUT. A resposta recebida é lida byte a byte e passada para a máquina de estado (função **updateStateMachine_CONNECTION()**) para a verificação. No lado do receptor, será aguardado pela trama de controlo SET, que também irá ser lida byte a byte e passada para a máquina de estado para a verificação, depois de receber a trama SET e verificar, é enviado ao emissor a trama UA como resposta. Assim que ambas as aplicações consigam estabelecer a ligação, esta função retorna com um valor de 0.

llclose() - Responsável de terminar a conexão entre o emissor e o receptor e também reconfigurar as definições da porta série para a definição inicial (que foi modificada dentro da função llopen() e guardada numa variável **oldtermios**). No emissor é enviada uma trama DISC e esperada outra trama DISC enviada do lado do receptor (lida byte a byte e verificado na função updateStateMachine_CLOSE()), no sucesso recepção da trama DISC, é finalizado com o envio da trama UA como resposta da trama DISC recebida e logo de seguida termina o programa. No receptor é esperado a trama DISC do emissor, ao receber e verificar com sucesso é enviada também uma trama DISC ao emissor, guardando pela trama UA de resposta durante um intervalo de tempo definido (TIME_OUT_SCS), se não receber com sucesso, este também terminará o programa mas dando um warning.

llread()- A função llread() está constantemente à espera para receber uma trama, quando recebe a função verifica primeiro se a trama é válida (caso não seja rejeita-a enviando um REJ) procedendo depois com o processo de verificação fazendo o destuffing ao pacote da trama e ao BCC que corresponde ao pacote e calcula-se o BCC2.Seguindo com a verificação se o pacote da trama é válido enviando uma trama RR mesmo que o pacote seja duplicado , caso não seja ,envia uma trama REJ.

llwrite()-A função llwrite() é responsável por fazer o stuffing e o envio das tramas.Para conseguir este fim é primeiro feito o framing da mensagem, ou seja, acrescentado o cabeçalho do Protocolo de Ligação à mensagem (para calcular o BCC2 é chamada a função bcc2). É feito também o stuffing da mensagem e do BCC2. A partir deste momento a trama está pronta para ser enviada. A escrita é feita trama a trama e o seu envio usa o mesmo mecanismo de timeout de retransmissão do sinal SET usado no llopen().

Protocolo de aplicação

O protocolo de aplicação começa primeiro por fazer parse do número da porta que foi entrada como argumento da linha de comando, e de seguida pedir que tipo de identidade a aplicação atual vai ser corrida (emissor ou receptor), no caso de ser o emissor, ainda vai ser pedido o nome do ficheiro a enviar. Estes dados são validados, em caso de sucesso, a aplicação começa por estabelecer a ligação entre o emissor e receptor executando a função llopen(). Depois de estabelecer a ligação, é começado a contagem do tempo que a aplicação demora a transmitir o ficheiro. No caso do emissor, é chamado a função sendPacket(), que começa por construir a trama de controlo, onde é guardado as informações sobre o ficheiro a enviar (tamanho e nome do ficheiro), e enviada com a função llwrite() (responsável por construir e enviar a trama I, e aguardar pela resposta do receptor), depois segue para um ciclo while que é feita a leitura de parte do ficheiro (cada leitura são lidas **DATA_MAX_SIZE - 4** bytes de vez), com esses dados é construído o pacote de dados e enviada para o emissor com a função llwrite() até terminar de enviar todo o ficheiro, que no fim envia ainda mais um pacote de controlo com flag de END para indicar o final da transferência do ficheiro ao receptor. No caso do receptor, é chamado a função receivePacket(), onde os dados são lidos através da função llread() (onde recebe as trama I, faz destuffing e a validação da trama) e depois processa os pacotes lidos dentro de um ciclo while, com a condição de paragem quando receber o pacote de controlo com a flag de END.No fim, a contagem de tempo é terminado e é executado a função llclose() para terminar a ligação entre o receptor e emissor, é executado a função logStats() que imprime o tempo consumido e o Bit Rate para a transmissão do ficheiro .

Validação

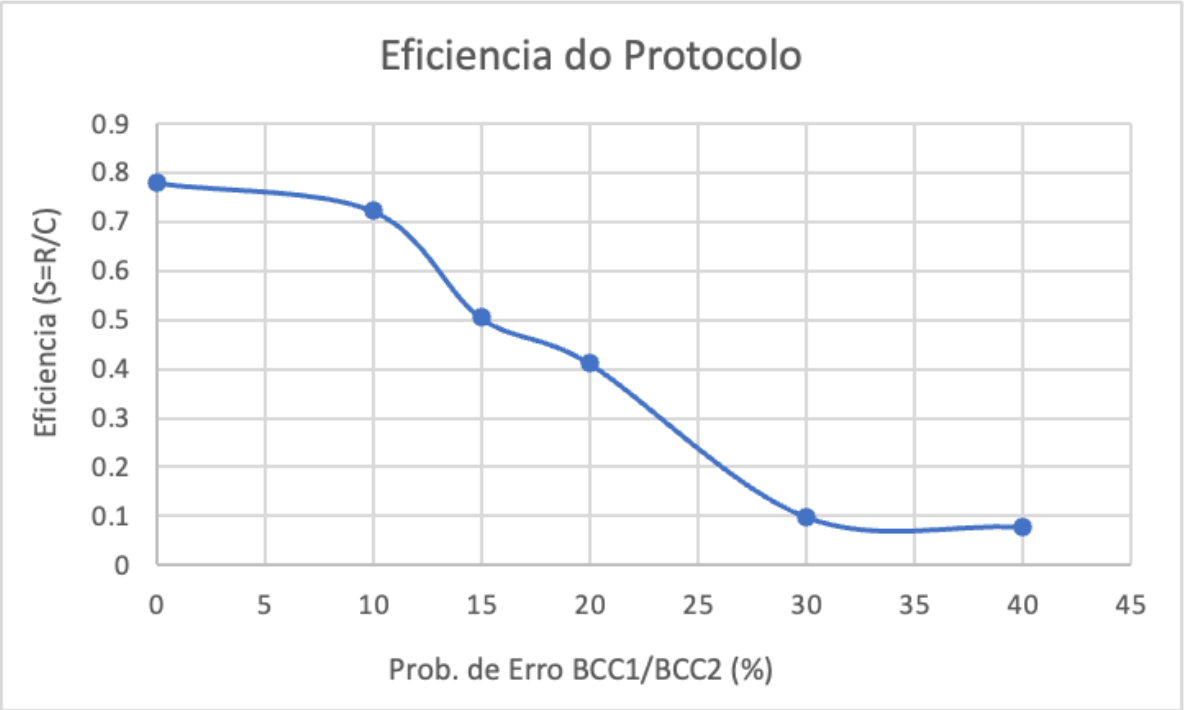
Para testar a nossa aplicação usamos diferentes tipos de ficheiros para garantir que a nossa aplicação abrange uma grande variedade de tipos de ficheiros e não perde informação quando se está a transferir um ficheiro maior. Testamos também com ficheiros de outros formatos, por exemplo jpg ,png e gif. Para efeitos de teste geramos erros durante a transmissão nos BCC1 e BCC2 com as funções generateErrorBCC1() e generateErrorBCC2(), e concluímos que a nossa aplicação funciona sem problemas. Na nossa aplicação também é possível usar diferentes valores de baudrate (capacidade de transmissão) e tamanhos de tramas I .

Eficiência do protocolo de ligação de dados

A eficiência do protocolo de ligação de dados foi obtido através de testes efetuados na aplicação, medindo o tempo que o programa demora a transmitir as tramas de Informações (desde a primeira trama I até a última trama I) com um ficheiro de 103,511 bytes (ou seja 828088 bits), este tempo foi medido ambo nos lados do emissor e receptor é calculado a média desse valor. Depois de saber o tempo consumido, o **Bit Rate (R)** é calculado por tamanho do ficheiro dividido pelo tempo. A **eficiência (S)** é calculada dividindo o **Bit Rate (R)** obtido por **Baudrate (C)** pré definido, com valor de 38400 bits/s).

A aplicação foi testada primeiro sem erros, depois foi aumentado a percentagem de erros BBC1 ou BBC2 com os seguintes valores apresentados nas tabelas em baixo. Olhando para os resultados obtidos, podemos concluir que o FER (Frame Error Ratio) tem um impacto grande na eficiência do protocolo. Isto porque, foi usado um mecanismo de stop and wait que precisa de uma resposta sempre que for enviada uma trama, no caso de ocorrer uma perda de resposta, ainda é preciso a espera de um intervalo de tempo para reenviar a trama.

Tamanho de Ficheiro (bits)	Tamanho da Trama (bytes)	Baudrate (C) (bits/segundos)	Prob. Erro BBC1/BCC2	Tempo Consumido (segundos)	Bit Rate (R) (bits/segundos)	Eficiencia (S = R/C)
828088	1024	38400	0%	27.655773	29942.68141	0.77975733
828088	1024	38400	10%	29.831826	27758.54217	0.7228787
828088	1024	38400	15%	42.724565	19382.01126	0.50473988
828088	1024	38400	20%	52.321648	15826.87151	0.41215811
828088	1024	38400	30%	217.195311	3812.642162	0.09928756
828088	1024	38400	40%	273.453093	3028.263425	0.07886103



Conclusão

Em síntese podemos concluir que o trabalho foi concluído com sucesso pois conseguimos desenvolver uma aplicação onde é possível enviar ficheiros de um computador para outro sem perdas usando uma porta série, mesmo gerando erros. Embora não conseguimos concluir se a nossa aplicação funciona quando se gera ruídos.

Com a elaboração deste 1º trabalho prático conseguimos consolidar o que foi aprendido tanto na parte prática como na parte teórica.

ANEXO - Código Fonte

Código também disponível no github:
(https://github.com/FabioMiguel2000/LEIC-Redes-de-Computadores-2021-22)

application.c

```
#include "application.h"



int parseArgs(int argc, char **argv)

{

   if(argc != 2){

       return -1;

   }

   if (strcmp("/dev/ttyS10", argv[1]) == 0)

   {

       return 10;

   }

   if (strcmp("/dev/ttyS11", argv[1]) == 0)

   {

       return 11;

   }

   if (strcmp("/dev/ttyS0", argv[1]) == 0)

   {

       return 0;

   }

   if (strcmp("/dev/ttyS1", argv[1]) == 0)

   {
```

```
        return 1;
    }
    return -1;
}


int getIdentity(){
    char buf[MAX_SIZE];
    do{
        logInfo("Choose identity:\n\t\t1.RECEIVER\n\t\t2.TRANSMITTER\n  ");
        fgets(buf, MAX_SIZE, stdin);
        if(buf[strlen(buf)-1] ==  '\n'){  //Removes newline from buffer
            buf[strlen(buf)-1] = '\0';
        }
        if(strcmp("1", buf) == 0 || strcmp("RECEIVER", buf) == 0){
            applicationLayer.status = RECEIVER;
            break;
        }
        if(strcmp("2", buf) == 0 || strcmp("TRANSMITTER", buf) == 0){
            do{
                logInfo("Input name of file to transmit\n");
                fgets(buf, MAX_SIZE, stdin);
                if(buf[strlen(buf)-1] ==  '\n'){  //Removes newline from buffer
                    buf[strlen(buf)-1] = '\0';
                }
                strcpy(dataFile.filename, buf);
                struct stat fileInfo;
```

```
            if(stat(dataFile.filename, &fileInfo) == 0){
                dataFile.filesize = fileInfo.st_size;
                break;
            }
            logWarning("No file found! Please try again!\n");
        } while (1);
        applicationLayer.status = TRANSMITTER;
        break;
    }
    logWarning("Invalid input! Please try again!\n");
} while (1);
return 0;
}


int sendPacket(int fd)
{
    char msg[MAX_SIZE];
    unsigned char controlPacket[MAX_SIZE];


    //  Build control packet
    controlPacket[0] = CTRL_PACK_C_START;
    controlPacket[1] = CTRL_PACK_T_SIZE;
    controlPacket[2] = sizeof(dataFile.filesize);
    memcpy(&controlPacket[3], &dataFile.filesize, sizeof(dataFile.filesize));


    controlPacket[3 + sizeof(dataFile.filesize)] = CTRL_PACK_T_NAME;
```

```
   controlPacket[4 + sizeof(dataFile.filesize)] = strlen(dataFile.filename);

   memcpy(&controlPacket[5 + sizeof(dataFile.filesize)], &dataFile.filename,
strlen(dataFile.filename));



   if(llwrite(fd, controlPacket, strlen(dataFile.filename) + 5 +
sizeof(dataFile.filesize))<0){

       logError("Something went wrong while sending START control packet on
llwrite()!\n");

       exit(-1);

   }

   logInfo("START control packet was transmitted!\n");



   int count = 0;



   int dataFileFd = open(dataFile.filename, O_RDONLY);



   if (dataFileFd < -1)

   {

       logError("Unable to open data file!\n");

   }



   unsigned char dataPacket[DATA_MAX_SIZE];

   unsigned char data[DATA_MAX_SIZE - 4];




   sprintf(msg, "File Information:\n\t\tFile name: %s\n\t\tFile total size: %ld
Bytes\n", dataFile.filename, dataFile.filesize);
```

```
    logInfo(msg);

    logInfo("Starting to send file data...\n");



    off_t sizeLeft = dataFile.filesize;

    int bytesRead = read(dataFileFd, &data, DATA_MAX_SIZE - 4);



    while (bytesRead > 0)

    {



        // printf("%i, bytes read, count num = %i\n", bytesRead, count);

        dataPacket[0] = CTRL_PACK_C_DATA;

        dataPacket[1] = count % 255;

        // K = l2*256 + l1, where l1 = dataPacket[2] & l2 = dataPacket[3]

        dataPacket[2] = bytesRead / 256;

        dataPacket[3] = bytesRead % 256;



        memcpy(&dataPacket[4], data, bytesRead);



        if(llwrite(fd, dataPacket, bytesRead + 4) < 0){

            logError("Something went wrong while sending file information on
llwrite()!\n");

            exit(-1);

        }

        sizeLeft -= bytesRead;

        sprintf(msg, "Transmission Number = %i\n\t\t> %i Bytes was transmitted on
this transmission\n\t\t> %li Bytes left!\n", count, bytesRead, sizeLeft);

        logInfo(msg);
```

```
        count++;

        bytesRead = read(dataFileFd, &data, DATA_MAX_SIZE - 4);

    }

    if(sizeLeft == 0){

        sprintf(msg, "All %li Bytes successfully transmitted!\n",
dataFile.filesize);

        logSuccess(msg);

    }

    else{

        sprintf(msg, "Only %li bytes transmitted, expected %li bytes!",
dataFile.filesize - sizeLeft, dataFile.filesize);

        logWarning(msg);

    }



    // Send End Control packet

    controlPacket[0] = CTRL_PACK_C_END;

    if(llwrite(fd, controlPacket, strlen(dataFile.filename) + 5 +
sizeof(dataFile.filesize))<0){

        logError("Something went wrong while sending END control packet on
llwrite()!\n");

        exit(-1);

    }



    logInfo("END control packet was transmitted!\n");



    return 0;

}
```

```
int receivePacket(int fd)

{

   unsigned char dataField[WORST_CASE_FRAME_I];

   char msg[MAX_SIZE];

   int bytesRead;

   int sequenceNum = 0;

   int stop = 0;

   off_t totalSizeLoaded = 0;

   while (!stop)

   {

       bytesRead = llread(fd, dataField);

       if (bytesRead < 0)

       {

           continue;

       }



       if (dataField[0] == CTRL_PACK_C_DATA)

       {

           int dataSize;

           if ((sequenceNum % 255) != dataField[1])

           {

               sprintf(msg, "Sequence number do not match, received = %i, actual =
%i\n", dataField[1], (sequenceNum % 255));

               logWarning(msg);

               continue;
```

```
            }

            dataSize = 256 * dataField[2] + dataField[3];

            if (dataFile.fd > 0)

            { //If already openned



                write(dataFile.fd, &dataField[4], dataSize);

                totalSizeLoaded += dataSize;

                sprintf(msg, "Transmission Number = %i\n\t\t> %i Bytes was received
on this transmission\n\t\t> %li Bytes total received!\n", sequenceNum, bytesRead,
totalSizeLoaded);

                logInfo(msg);



                sequenceNum++;



            }

        }

        else if (dataField[0] == CTRL_PACK_C_START)

        {

            logInfo("START control packet was received!\n");

            int V_fieldSize;

            int bytesProccessed = 1;

            while (bytesProccessed < bytesRead)

            {

                if (dataField[bytesProccessed] == CTRL_PACK_T_SIZE)

                {

                    V_fieldSize = dataField[bytesProccessed + 1];

                    for (int i = 0; i < V_fieldSize; i++)
```

```
                {
                    dataFile.filesize += dataField[bytesProccessed + 2 + i] << 8
* i;
                }

                bytesProccessed += V_fieldSize + 1;
            }
            else if (dataField[bytesProccessed] == CTRL_PACK_T_NAME)
            {
                V_fieldSize = dataField[bytesProccessed + 1];
                for (int i = 0; i < V_fieldSize; i++)
                {
                    dataFile.filename[i] = dataField[bytesProccessed + 2 + i];

                }
                bytesProccessed += V_fieldSize + 1;
            }
            bytesProccessed ++;
        }
        char newFilename[MAX_SIZE+5];
        sprintf(newFilename, "copy_%s", dataFile.filename);
        dataFile.fd = open(newFilename, O_RDWR | O_CREAT, 0777);        //0777
for permission
        if (dataFile.fd < 0)
        {
            logError("Unable to open file to load data!\n");
            return -1;
```

```
            }

            sprintf(msg, "File Information:\n\t\tFile name: %s\n\t\tFile total size:
%ld Bytes\n", dataFile.filename, dataFile.filesize);

            logInfo(msg);

        }

        else if (dataField[0] == CTRL_PACK_C_END)

        {

            logInfo("END control packet was received!\n");

            stop = 1;

            if(close(dataFile.fd)<0){

                printf("Error closing the copy file!\n");

            }



            if (totalSizeLoaded == dataFile.filesize)

            {

                sprintf(msg, "All %li bytes successfully received!\n",
totalSizeLoaded);

                logSuccess(msg);

            }

            else

            {

                sprintf(msg, "Only %li bytes received, expected %li bytes!\n",
totalSizeLoaded, dataFile.filesize);

                logWarning(msg);

            }

        }

    }
```

```
    return 0;
}

int main(int argc, char **argv)

{

    int portNum = parseArgs(argc, argv);

    if (portNum < 0)
    {
        logUsage();
        exit(-1);
    }
    if(getIdentity() < 0){
        logError("Unable to get application identity!\n");
        exit(-1);
    }
    int fd = llopen(portNum, applicationLayer.status);
    if (fd < 0)
    {
        exit(-1);
    }
    startTimeElapsed();
    switch (applicationLayer.status)
    {
```

```
    case TRANSMITTER:

        IDENTITY = TRANSMITTER;

        if (sendPacket(fd) < -1)

        {

            exit(-1);

        }

        break;

    case RECEIVER:

        IDENTITY = RECEIVER;

        if(receivePacket(fd) < -1){

            exit(-1);

        }

        break;

    default:

        break;

    }

    endTimeElapsed();

    llclose(fd);

    logStats();

    return 0;

}
```

application.h

```
#ifndef APPLICATION_H
#define APPLICATION_H
```

```c
#include "macros.h"
#include "utils.h"
#include "stateMachine.h"
#include "protocol.h"


int IDENTITY;

struct applicationLayer applicationLayer;

struct dataFile dataFile;

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

struct dataFile {
   char filename[100];
   off_t filesize;
   int fd;
};

/**
* @brief Parses app arguments
*
* @param argc      Argument count
* @param arg      Argument vector
* @return int     Port number that has been passed by user, -1 if error
*/
int parseArgs(int argc, char** arg);

/**
* @brief Expect UA to arrive from port.
*
* @param fd        file descriptor for serial port connection
* @return int      On success, 0; -1 otherwise
*/
int receiver_UA(int fd);
/**
* @brief Transmitter function that handles the file data and sends it to the
receiver.
*
* @param fd        file descriptor for serial port connection
* @return int      0 on success, -1 otherwise
```

```
*/
int transmitter_SET(int fd);
/**
* @brief Receiver function that receives and process the file data comming from the
transmitter.
*
* @param fd          file descriptor for serial port connection
* @return int        0 on success, -1 on error
*/
int sendPacket(int fd);

/**
* @brief Gets the identity that current application (receiver or transmitter), in
case of transmitter, will also ask for file name to be send.
* @return int     return 0 if success
*/
int getIdentity();
/**
* @brief Main application function.
*
* @param argc          Arguments count
* @param arg           Argument vector
* @return int          0 on success, -1 on error
*/
int main(int argc, char** arg);


#endif
```

protocol.c

```
#include "protocol.h"

extern int timeout, timeoutCount;
extern int frameISize;
struct termios oldtio;

int receiver_UA(int fd)
{
   int res;
   unsigned char buf[MAX_SIZE];

   stateMachine_st stateMachine;
```

```
    stateMachine.currState = START;

    logInfo("Receiver waiting to set connection with transmitter...\n");

    while (stateMachine.currState != STOP)
    {                           /* loop for input */
        res = read(fd, buf, 1); /* returns after 1 char have been input */
        buf[res] = 0;           /* so we can printf... */
        if (res != -1)
        {
            updateStateMachine_CONNECTION(&stateMachine, buf);
        }
    }

    buf[0] = FLAG;
    buf[1] = A_CERR;
    buf[2] = C_UA;
    buf[3] = BCC(A_CERR, C_UA);
    buf[4] = FLAG;

    res = write(fd, buf, 5); //Sends UA to the sender

    logSuccess("Connection with transmitter was sucessfully established!\n");

    return 0;
}

int transmitter_SET(int fd)
{
    int res;
    unsigned char buf[MAX_SIZE];

    buf[0] = FLAG;
    buf[1] = A_CERR;
    buf[2] = C_SET;
    buf[3] = BCC(A_CERR, C_SET); //Will be trated as null character (\0) if = 0x00
    buf[4] = FLAG;

    signal(SIGALRM, timeoutHandler);

    timeout = 0;
    timeoutCount = 0;
    stateMachine_st stateMachine;
    stateMachine.currState = START;
```

```
    alarm(TIME_OUT_SCS);     // 3 seconds timout
    res = write(fd, buf, 5); //Sends the data to the receiver
    if (res < 0)
    {
        logError("Unable to send SET to receiver on function transmitter_SET()!\n");
        exit(-1);
    }

    logInfo("SET sent! Transmitter trying to establish connection with
receiver...\n");

    while (stateMachine.currState != STOP)
    { /* loop for input */
        if (timeout)
        {
            if (timeoutCount >= MAX_TRIES)
            {
                logError("TIMEOUT, UA not received!\n");
                exit(-1);
            }
            res = write(fd, buf, 5); //SENDS DATA TO RECEIVER AGAIN
            timeout = 0;
            stateMachine.currState = START;
            alarm(TIME_OUT_SCS);
        }

        res = read(fd, buf, 1); /* returns after 1 char have been input */
        buf[res] = 0;           /* so we can printf... */

        if (res == 1){
            updateStateMachine_CONNECTION(&stateMachine, buf);
        }
    }
    logSuccess("UA received! Connection with receiver was sucessfully
established!\n");

    return 0;
}

int llopen(int portNum, int identity)
{
    int fd;

    //  initiate linkLayer struct
    sprintf(linkLayer.port, "/dev/ttyS%i", portNum);
```

```
linkLayer.baudRate = BAUDRATE;
linkLayer.sequenceNumber = identity == TRANSMITTER ? 0 : 1;
linkLayer.timeout = TIME_OUT_SCS;
linkLayer.numTransmissions = TIME_OUT_CHANCES;

struct termios newtio;
fd = open(linkLayer.port, O_RDWR | O_NOCTTY | O_NONBLOCK);
if (fd < 0)
{
    logError("Function llopen(), could not open port!\n");
    return -1;
}

if (tcgetattr(fd, &oldtio) == -1)
{ /* save current port settings */
    logError("Function llopen(), error on tcgetattr()!\n");
    return -1;
}

bzero(&newtio, sizeof(newtio));
newtio.c_cflag = linkLayer.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; /* inter-character timer unused */
newtio.c_cc[VMIN] = 5;  /* blocking read until 5 chars received */

/* VTIME e VMIN devem ser alterados de forma a proteger com um temporizador a
leitura do(s) pr@ximo(s) caracter(es)*/
tcflush(fd, TCIOFLUSH);

if (tcsetattr(fd, TCSANOW, &newtio) == -1)
{
    logError("Function llopen(), error on tcgetattr()!\n");
    return -1;
}

logSuccess("New termios structure set\n");

switch (identity)
{
case TRANSMITTER:
```

```
        if (transmitter_SET(fd) < 0)
        {
            logError("Unable to establish connection with receiver at function
llopen()\n");
            return -1;
        }
        break;
    case RECEIVER:
        if (receiver_UA(fd) < 0)
        {
            logError("Unable to establish connection with transmitter at function
llopen()\n");
            return -1;
        }
        break;
    default:
        logError("Unable to establish connection in function llopen(), incorrect
identity\n");
        return -1;
        break;
    }
    return fd;
}

int llclose(int fd){
    unsigned char *frame = (unsigned char *)malloc(5);
    unsigned char buf;
    timeout = 0;
    stateMachine_st stateMachine;
    int res;
    int numtries = 0;
    signal(SIGALRM, timeoutHandler);
    switch (IDENTITY)
    {
    case TRANSMITTER:
        frame[0] = FLAG;
        frame[1] = A_CERR;
        frame[2] = C_DISC;
        frame[3] = BCC(A_CERR, C_DISC); //envia trama de comando DISC
        frame[4] = FLAG;
        res = write(fd, frame, 5);
        if (res < 0){
            logError("Unable to send DISC to receiver on function llclose()!\n");
            return -1;
        }
```

```
        logInfo("DISC frame was sent, trying to disconnect with receiver.\n");

        //state machine -aguarda disc do receiver
        stateMachine.currState = START;
        stateMachine.A_Expected = A_CRRE; //aqui
        stateMachine.C_Expected = C_DISC;
        //AGUARDA RESPOSTA TRAMA DE COMANDO DISC COMO RESPOSTA DO RECEIVER
        alarm(TIME_OUT_SCS); // 3 seconds timout
        while (stateMachine.currState != STOP)
        { /* loop for input */
            if (timeout)
            {
                numtries++;
                if (numtries >= MAX_TRIES)
                {
                    logError("TIMEOUT, DISC not received from received!\n");
                    return -1;
                }
                res = write(fd, frame, 5); //SENDS DATA TO RECEIVER AGAIN
                if (res < 0){
                    logError("Could not write DISC to receiver, on function
llclose()!\n");
                    return -1;
                }
                timeout = 0;
                stateMachine.currState = START;
                alarm(TIME_OUT_SCS);
            }
            res = read(fd, &buf, 1);
            if(res == 1){
                updateStateMachinell_CLOSE(&stateMachine, &buf, IDENTITY);
            }
        }
        logInfo("DISC frame received from receiver.\n");
        //ENVIA TRAMA DE COMANDO UA
        frame[0] = FLAG;
        frame[1] = A_CRRE;
        frame[2] = C_UA;
        frame[3] = BCC(A_CRRE, C_UA);
        frame[4] = FLAG;
        res = write(fd, frame, 5);
        logInfo("UA frame sent to receiver, ready to disconnect.\n");
        if (res < 0)
        {
            logError("Could not respond UA to receiver, on function llclose()!\n");
```

```
            return -1;
        }
        break;

    case RECEIVER:
        //STATE MACHINE AGUARDA DISC DO TRANSMITTER
        stateMachine.currState = START;
        stateMachine.A_Expected = A_CERR;
        stateMachine.C_Expected = C_DISC;
        //AGUARDA TRAMA (DISC) COMO COMANDO ENVIADO PELO EMISSOR
        while (stateMachine.currState != STOP)
        {
            res = read(fd, &buf, 1);
            if(res == 1){
                updateStateMachinell_CLOSE(&stateMachine, &buf, IDENTITY);
            }
        }
        logInfo("DISC frame received, transmitter asking to disconnect.\n");
        frame[0] = FLAG;
        frame[1] = A_CRRE;
        frame[2] = C_DISC;
        frame[3] = BCC(A_CRRE, C_DISC);
        frame[4] = FLAG;
        res = write(fd, frame, 5); //Envia Comando (DISC)
        if (res < 0)
        {
            logError("Could not write DISC to transmitter, on function
llclose()!\n");
            return -1;
        }
        logInfo("DISC frame was sent, waiting for UA response from transmitter.\n");

        //STATE MACHINE AGUARDA UA DO TRANSMITTER
        stateMachine.currState = START;
        stateMachine.A_Expected = A_CRRE;
        stateMachine.C_Expected = C_UA;
        signal(SIGALRM, disconnectTimeout);
        alarm(TIME_OUT_SCS);
        while (stateMachine.currState != STOP)
        { /* loop for input */
            res = read(fd, &buf, 1); /* returns after 1 char have been input */
            if(res == 1){
                updateStateMachinell_CLOSE(&stateMachine, &buf, IDENTITY);
            }
            if(timeout){
```

```
                break;
            }
        }
        if(timeout){
            timeout = 0;
            logWarning("UA was not responded, ready to disconnect.\n");
        }
        else{
            logInfo("UA frame received, ready to disconnect.\n");
        }

        break;
    default:
        break;
    }

    if (tcsetattr(fd, TCSANOW, &oldtio) == -1)
    {
        logError("Function llclose(), error on tcsetattr\n!");
        return -1;
    }
    close(fd);
    logSuccess("Application successfully terminated!\n");
    return 0;
}

int llwrite(int fd, unsigned char *dataField, int dataLength)
{

    //-----Calculate BCC2, using the data field before stuffing-------
    unsigned char BCC2 = dataField[0];
    for (int i = 1; i < dataLength; i++)
    {
        BCC2 ^= dataField[i];
    }
    //-----------------------------------------------------------------

    //-----------------Data Field Stuffing--------------------------------
    unsigned char *stuffedDataField = (unsigned char *)malloc(dataLength); //Data
field after stuffing
    int stuffedDataLength = dataLength;                                    //Size of
dataField after stuffing

    stuffedDataField[0] = dataField[0];
    int stuffed_index = 0;
```

```
    for (int i = 0; i < dataLength; i++)
    {
        if (dataField[i] == FLAG)
        {
            stuffedDataLength++;
            stuffedDataField = (unsigned char *)realloc(stuffedDataField,
stuffedDataLength);
            stuffedDataField[stuffed_index] = ESCAPE;
            stuffedDataField[stuffed_index + 1] = FLAG_ESC;
            stuffed_index += 2;
        }
        else if (dataField[i] == ESCAPE)
        {
            stuffedDataLength++;
            stuffedDataField = (unsigned char *)realloc(stuffedDataField,
stuffedDataLength);
            stuffedDataField[stuffed_index] = ESCAPE;
            stuffedDataField[stuffed_index + 1] = ESC_ESC;
            stuffed_index += 2;
        }
        else
        {
            stuffedDataField[stuffed_index] = dataField[i];
            stuffed_index += 1;
        }
    }

    //-----------------------------------------------------------------

    //---------------BCC2 stuffing----------------------
    unsigned char *stuffedBCC2 = (unsigned char *)malloc(1);
    int BCC2Length = 1;
    if (BCC2 == FLAG)
    {
        stuffedBCC2 = (unsigned char *)realloc(stuffedBCC2, 2);
        stuffedBCC2[0] = ESCAPE;
        stuffedBCC2[1] = FLAG_ESC;
        BCC2Length = 2;
    }
    else if (BCC2 == ESCAPE)
    {
        stuffedBCC2 = (unsigned char *)realloc(stuffedBCC2, 2);
        stuffedBCC2[0] = ESCAPE;
        stuffedBCC2[1] = ESC_ESC;
        BCC2Length = 2;
```

```
    }
    else
    {
        stuffedBCC2[0] = BCC2;
    }
    //-------------------------------------------------

    //----------------Building Frame I---------------------------------
    int frameISize = 5 + BCC2Length + stuffedDataLength; //Size of frame I
    unsigned char frameI[frameISize];                    //Allocate memory with size
of frame I calculated
    frameI[0] = FLAG;
    frameI[1] = A_CERR;
    frameI[2] = C_I(linkLayer.sequenceNumber);
    frameI[3] = BCC(A_CERR, C_I(linkLayer.sequenceNumber));
    memcpy(&frameI[4], stuffedDataField, stuffedDataLength);
    memcpy(&frameI[4 + stuffedDataLength], stuffedBCC2, BCC2Length);
    frameI[4 + stuffedDataLength + BCC2Length] = FLAG;

    //-----------------------------------------------------------------
    signal(SIGALRM, timeoutHandler);

    timeout = 0;
    timeoutCount = 0;
    stateMachine_st stateMachine;
    stateMachine.currState = START;

    unsigned char response[MAX_SIZE];

    alarm(TIME_OUT_SCS); // set alarm, 3 seconds timout

    unsigned char frameIProbError[frameISize];  //This frame maybe changed to a
frame with error
    memcpy(&frameIProbError[0], frameI, frameISize);

    generateErrorBCC1(frameIProbError);
    generateErrorBCC2(frameIProbError, frameISize, BCC2Length);

    int res = write(fd, frameIProbError, frameISize); //Sends the frame I (that may
contain an error) to the receiver
    if (res < 0)
    {
        logError("Unable to write frame I to receiver!\n");
        exit(-1);
    }
```

```
   int status = 0;
   //  Waits for response (RR or REJ from receiver), and resends the frame if
needed
   while (stateMachine.currState != STOP)
   { /* loop for input */
       if (timeout)
       {
           if (timeoutCount >= 3)
           {
               logError("TIMEOUT, UA not received!\n");
               exit(-1);
           }
           res = write(fd, frameI, frameISize); //SENDS DATA TO RECEIVER AGAIN
           logWarning("Resending Frame, due to TIMEOUT!\n");
           timeout = 0;
           stateMachine.currState = START;
           alarm(TIME_OUT_SCS);
       }
       else if(status == -1){  //Rejected

           res = write(fd, frameI, frameISize); //REJECTED, SENDS DATA TO RECEIVER
AGAIN
           logWarning("Resending Frame, due to REJECT!\n");

           stateMachine.currState = START;
       }

       res = read(fd, response, 1); /* returns after 1 char have been input */
       response[res] = 0;           /* so we can printf... */

       if (res != -1)
       {

           status = updateStateMachine_COMMUNICATION(&stateMachine, response);
       }
       //stateMachine.currState = STOP;
   }
   linkLayer.sequenceNumber = (linkLayer.sequenceNumber + 1) % 2;
   return frameISize;
}

//buffer -> Data field stored in the frame I, which has a maximum size of
DATA_MAX_SIZE
int llread(int fd, unsigned char *buffer)
{
```

```
   int res;
   unsigned char buf[MAX_SIZE];
   int machineState;
   char response[MAX_SIZE];
   stateMachine_st stateMachine;
   stateMachine.currState = START;

   response[0] = FLAG;
   response[1] = A_CERR;
   response[4] = FLAG;
   while (stateMachine.currState != STOP)
   {                           /* loop for input */
       res = read(fd, buf, 1); /* returns after 1 char have been input */
       buf[res] = 0;           /* so we can printf... */

       if (res != -1)
       {

           machineState = updateStateMachine_COMMUNICATION(&stateMachine, buf);
           if (machineState == INCORRECT_C_FIELD || machineState ==
INCORRECT_BCC1_FIELD)
           {
               response[2] = C_REJ((linkLayer.sequenceNumber + 1) % 2);
               response[3] = BCC(A_CERR, C_REJ((linkLayer.sequenceNumber + 1) %
2));

               res = write(fd, response, 5);
               printf("Reject sent!\n");
               if (res < 0)
               {
                   logError("Unable to send REJ to transmitter!\n");
               }
           }
       }
   }

   //fazer destuffing ao linklayer.frame
   unsigned char destuffedBCC2;
   int stuffedBCC2Size = 2;
   //BCC2 destuffing
   if (linkLayer.frame[frameISize - 3] == ESCAPE && linkLayer.frame[frameISize - 2]
== ESC_ESC)
   {
       destuffedBCC2 = ESCAPE;
   }
```

```
    else if (linkLayer.frame[frameISize - 3] == ESCAPE && linkLayer.frame[frameISize
- 3] == FLAG_ESC)
    {
        destuffedBCC2 = FLAG;
    }
    else
    {
        destuffedBCC2 = linkLayer.frame[frameISize - 2];
        stuffedBCC2Size = 1;
    }
    //Data destuffing and calculate actual BCC2
    int stuffedDataSize = frameISize - 5 - stuffedBCC2Size;
    int destuffedDataSize = 0;
    unsigned char expectedBCC2 = 0x00;
    for (int i = 4; i < stuffedDataSize + 4; i++)
    {
        if (linkLayer.frame[i] == ESCAPE)
        {
            if (linkLayer.frame[i + 1] == FLAG_ESC)
            {
                buffer[destuffedDataSize] = FLAG;
            }
            else if (linkLayer.frame[i + 1] == ESC_ESC)
            {
                buffer[destuffedDataSize] = ESCAPE;
            }
            i++;
        }
        else
        {
            buffer[destuffedDataSize] = linkLayer.frame[i];
        }
        expectedBCC2 ^= buffer[destuffedDataSize];
        destuffedDataSize++;
    }
    if (expectedBCC2 != destuffedBCC2)
    {
        logError("Incorrect BCC2 received from receiver!\n");
        response[2] = C_REJ((linkLayer.sequenceNumber + 1) % 2);
        response[3] = BCC(A_CERR, C_REJ((linkLayer.sequenceNumber + 1) % 2));
        res = write(fd, response, 5);
        if (res < 0)
        {
            logError("Unable to send REJ to transmitter!\n");
            return -1;
```

```
        }
        return -1;
    }


    response[2] = C_RR(linkLayer.sequenceNumber);
    response[3] = BCC(A_CERR, C_RR(linkLayer.sequenceNumber));
    write(fd, response, 5);
    linkLayer.sequenceNumber = (linkLayer.sequenceNumber + 1) % 2;
    return destuffedDataSize;
}
```

protocol.h

```
#ifndef PROTOCOL_H
#define PROTOCOL_H

#include "macros.h"
#include "utils.h"
#include "stateMachine.h"
#include "application.h"

extern int IDENTITY;

struct linkLayer linkLayer;
/**
* @brief LinkLayer configuration structure.
*/
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*/
    char frame[WORST_CASE_FRAME_I];           /*Trama*/
};
/**
* @brief Opens a data connection with the serial port
*
* @param porta number of the port x in  "/dev/ttySx"
* @param type TRANSMITTER|RECEIVER
* @return int idata connection identifier or -1 in case of error
*/
int llopen(int portNum, int identity);
```

```
/**
* @brief Writes to serial port
*
* @param fd Port to where info will be written
* @param dataField Data that wil be written on port
* @param dataLength Length of data to be written on port
* @return number of written characters, negative otherwise.
*/
int llwrite(int fd, unsigned char *dataField, int dataLength);

/**
* @brief Read Serial Port
*
* @param fd Port to read
* @param buffer Data to be read
* @return Number of character that have been read, negative if error
*/
int llread(int fd, unsigned char *buffer);

/**
* @brief Close Serial Port
*
* @param fd Port to close
* @return int 0 if sucessful,negative otherwise
*/
int llclose(int fd);
#endif
```

statemachine.c

```
#include "stateMachine.h"

int frameISize;
extern struct linkLayer linkLayer;
extern struct applicationLayer applicationLayer;

void updateStateMachine_CONNECTION(stateMachine_st *currStateMachine, unsigned char
*buf){
   switch(currStateMachine->currState){
       case START:
           if(buf[0] == FLAG){
               currStateMachine->currState = FLAG_RCV;
           }
           break;
```

```
        case FLAG_RCV:
            if(buf[0] == A_CERR){
                currStateMachine->currState = A_RCV;
                currStateMachine->A_field = buf[0];
            }
            else if(buf[0] != FLAG){
                currStateMachine->currState = START;
            }
            break;
        case A_RCV:
            if((applicationLayer.status == RECEIVER && buf[0] == C_SET) ||
(applicationLayer.status == SENDER && buf[0] == C_UA)){
                currStateMachine->currState = C_RCV;
                currStateMachine->C_field = buf[0];
            }
            else if(buf[0] == FLAG){
                currStateMachine->currState = FLAG_RCV;
            }
            else if(buf[0] != FLAG){
                currStateMachine->currState = START;
            }
            break;
        case C_RCV:
            if(buf[0] == (currStateMachine->A_field ^ currStateMachine->C_field)){
//Check BCC
                currStateMachine->currState = BCC1_OK;
            }
            else if(buf[0] == FLAG){
                currStateMachine->currState = FLAG_RCV;

            }
            else if(buf[0] != FLAG){
                currStateMachine->currState = START;
            }
            break;
        case BCC1_OK:
            if(buf[0] == FLAG){
                currStateMachine->currState = STOP;
            }
            else if(buf[0] != FLAG){
                currStateMachine->currState = START;
            }
            break;
        default:
            break;
```

```
    }
}


int updateStateMachine_COMMUNICATION(stateMachine_st *currStateMachine, unsigned
char *buf){
    switch(currStateMachine->currState){
        case START:
            if(buf[0] == FLAG){
                currStateMachine->currState = FLAG_RCV;
                if(applicationLayer.status == RECEIVER){
                    frameISize = 0;
                    linkLayer.frame[frameISize] = buf[0];
                    frameISize ++;
                }

            }
            break;
        case FLAG_RCV:
            if(buf[0] == A_CERR){
                currStateMachine->currState = A_RCV;
                currStateMachine->A_field = buf[0];
                if(applicationLayer.status == RECEIVER){
                    linkLayer.frame[frameISize] = buf[0];
                    frameISize ++;
                }
            }
            else if(buf[0] == FLAG){
                currStateMachine->currState = FLAG_RCV;
                if(applicationLayer.status == RECEIVER){
                    frameISize = 1;
                    linkLayer.frame[0] = buf[0];
                }
            }
            else {
                currStateMachine->currState = START;
                if(applicationLayer.status == RECEIVER){
                    frameISize = 0;
                }
            }
            break;
        case A_RCV:
            if(applicationLayer.status == TRANSMITTER){
                if(buf[0] == C_RR((linkLayer.sequenceNumber+1) % 2)){
                    currStateMachine->currState = C_RCV;
```

```
                currStateMachine->C_field = buf[0];

            }
            else if(buf[0] == C_REJ(linkLayer.sequenceNumber)){
                logError("REJ received! Frame was rejected by receiver!\n");
                currStateMachine->currState = START;
                return -1;
            }
            else if(buf[0] == FLAG){
                currStateMachine->currState = FLAG_RCV;
            }
            else if(buf[0] != FLAG){
                currStateMachine->currState = START;
            }
        }
        else{   //identity == RECEIVER
            if(buf[0] == C_I((linkLayer.sequenceNumber+1) % 2)){
                currStateMachine->currState = C_RCV;
                currStateMachine->C_field = buf[0];
                linkLayer.frame[frameISize] = buf[0];
                frameISize ++;
            }
            else if(buf[0] != C_I((linkLayer.sequenceNumber+1) % 2)){
                currStateMachine->currState = START;
                frameISize = 0;
                logError("Incorrect Control Field received from transmitter!\n
");

                return INCORRECT_C_FIELD;
            }
            else if(buf[0] == FLAG){
                currStateMachine->currState = FLAG_RCV;
                linkLayer.frame[0] = buf[0];
                frameISize = 1;
            }
            else if(buf[0] != FLAG){
                currStateMachine->currState = START;
                frameISize = 0;
            }
        }
        break;
    case C_RCV:
        if(buf[0] == (currStateMachine->A_field ^ currStateMachine->C_field)){
//Check BCC
            currStateMachine->currState = BCC1_OK;
            if(applicationLayer.status == RECEIVER){
```

```
                linkLayer.frame[frameISize] = buf[0];
                frameISize ++;
            }
        }
        else if(buf[0] != (currStateMachine->A_field ^
currStateMachine->C_field)){
            if(applicationLayer.status == RECEIVER){
                currStateMachine->currState = START;
                frameISize = 0;
                logError("Incorrect BCC1 received from transmitter!\n ");
                return INCORRECT_BCC1_FIELD;
            }
        }
        else if(buf[0] == FLAG){
            currStateMachine->currState = FLAG_RCV;
            linkLayer.frame[0] = buf[0];
            frameISize = 1;
        }
        else if(buf[0] != FLAG){
            currStateMachine->currState = START;
            frameISize = 0;
        }
        break;
    case BCC1_OK:
        if(applicationLayer.status == TRANSMITTER){
            if(buf[0] == FLAG){
                currStateMachine->currState = STOP;
            }
            else{
                currStateMachine->currState = START;
            }
        }
        else{   //identity == RECEIVER
            currStateMachine->currState = INFO;
            linkLayer.frame[frameISize] = buf[0];
            frameISize ++;
        }
        break;
    case INFO:  //Can only be reached by receiver
        linkLayer.frame[frameISize] = buf[0];
        frameISize ++;
        if(buf[0] == FLAG){
            currStateMachine->currState = STOP;

        }
```

```
        case STOP:
            break;
        default:
            break;

    }
    return 0;
}

void updateStateMachinell_CLOSE(stateMachine_st *currStateMachine, unsigned char
*buf, int identity){
     switch(currStateMachine->currState){
        case START:

            if(*buf== FLAG){
                currStateMachine->currState = FLAG_RCV;
            }
            break;
        case FLAG_RCV:
            if(*buf== currStateMachine->A_Expected){
                currStateMachine->currState = A_RCV;
                currStateMachine->A_field = *buf;
            }
            else if(*buf != FLAG){

                currStateMachine->currState = START;
            }
            break;
        case A_RCV:
            if(*buf== currStateMachine->C_Expected){
                currStateMachine->currState = C_RCV;
                currStateMachine->C_field = *buf;
            }
            else if(*buf == FLAG){
                currStateMachine->currState = FLAG_RCV;
            }
            else if(*buf!= FLAG){
                currStateMachine->currState = START;
            }
            break;
        case C_RCV:
            if(*buf== BCC(currStateMachine->A_field,currStateMachine->C_field)){
//Check BCC
                currStateMachine->currState = BCC1_OK;
            }
```

```
            else if(*buf== FLAG){
                currStateMachine->currState = FLAG_RCV;

            }
            else if(*buf!= FLAG){
                currStateMachine->currState = START;
            }
            break;
        case BCC1_OK:
            if(*buf== FLAG){
                currStateMachine->currState = STOP;
            }
            else if(*buf!= FLAG){
                currStateMachine->currState = START;
            }
            break;
        default:
            break;
    }
}
```

statemachine.h

```
#ifndef STATEMACHINE_H
#define STATEMACHINE_H

#include "macros.h"
#include "utils.h"
#include "application.h"
#include "protocol.h"


enum stateMachine { START, FLAG_RCV, A_RCV, C_RCV, BCC1_OK, INFO, STOP};

typedef struct {
    enum stateMachine currState;
    unsigned char A_field;
    unsigned char C_field;
    char A_Expected;    //valor esperado no campo A (Command) em função do comando
que está a ser recebido
    char C_Expected;    //valor esperado no campo C (Command) em função do comando
que está a ser recebido
}stateMachine_st;
```

```
/**
* @brief Update state machine according to received byte to establish connection.
*
* @param currStateMachine   Pointer to state machine
* @param buf      Byte received, will decide the transition
* @return int      0 on success, other value otherwise
*/

void updateStateMachine_CONNECTION(stateMachine_st *currStateMachine, unsigned char
*buf);
/**
* @brief Update state machine according to received byte for packet exchange
between receiver and transmitter.
*
* @param currStateMachine   Pointer to state machine
* @param buf      Byte received, will decide the transition
* @return int      0 on success, other value otherwise
*/
int updateStateMachine_COMMUNICATION(stateMachine_st *currStateMachine, unsigned
char *buf);

/**
* @brief Update state machine according to received byte to be used in llclose in
order to close the connection.
*
* @param currStateMachine   Pointer to state machine
* @param buf      Byte received, will decide the transition
* @param identity Reference if it is TRANSMITER or RECEIVER
* @return int      0 on success, other value otherwise
*/
void updateStateMachinell_CLOSE(stateMachine_st *currStateMachine, unsigned char
*buf, int identity);
#endif
```

utils.c

```
#include "utils.h"

void logError(char *msg){
   char buf[MAX_SIZE];
   sprintf(buf, "\033[0;31m>>>ERROR:\t%s\n\033[0m", msg);
   write(STDOUT_FILENO, buf, strlen(buf));
}
```

```
void logSuccess(char *msg){
   char buf[MAX_SIZE];
   sprintf(buf, "\033[0;32m>>>SUCCESS:\t%s\n\033[0m", msg);
   write(STDOUT_FILENO, buf, strlen(buf));
}

void logWarning(char *msg){
   char buf[MAX_SIZE];
   sprintf(buf, "\033[0;33m>>>WARNING:\t%s\n\033[0m", msg);
   write(STDOUT_FILENO, buf, strlen(buf));
}

void logInfo(char *msg){
   char buf[MAX_SIZE];
   sprintf(buf, ">>>Info:\t%s\n", msg);
   write(STDOUT_FILENO, buf, strlen(buf));
}

void logUsage(){
   char buf[MAX_SIZE];
   sprintf(buf, "Usage:\t./application <SerialPort>\n\tex: ./application
/dev/ttyS10\n");
   write(STDOUT_FILENO, buf, strlen(buf));

}

void timeoutHandler()                    // atende alarme
{
 timeout=1;
 timeoutCount++;
 char buf[MAX_SIZE];
 sprintf(buf, "Time-out achieved, count = %i\n", timeoutCount);
 logWarning(buf);

}

void disconnectTimeout()                    // atende alarme
{
 timeout=1;
}


void startTimeElapsed(){
   timeElapsed.start = clock();
}
```

```
void endTimeElapsed(){
   timeElapsed.end = clock();
   timeElapsed.timeTaken = ((double) (timeElapsed.end - timeElapsed.start)) /
CLOCKS_PER_SEC;
}

void logStats(){
   char buf[MAX_SIZE];
   sprintf(buf, "Transmission Time: %f s\nTransmission Rate: %f Bytes/s\n",
timeElapsed.timeTaken, dataFile.filesize/timeElapsed.timeTaken);
   write(STDOUT_FILENO, buf, strlen(buf));
}

void generateErrorBCC2(unsigned char *frame, int size, int stuffedBCC2Size){
 int errorFlag = (rand() % 100) < ERROR_PROBABILITY_BCC2;
 if (errorFlag){
   char buf[MAX_SIZE];
   int index = (rand() % (size - 1 - stuffedBCC2Size)) + 4;  //Index only in data
field range
   frame[index] = frame[index] ^ 0xff;         //Negates a byte
   sprintf(buf,"Generated BCC2 with errors.\n\t\t> Frame at index: %i = %#x =>
%#x\n", index, frame[index] ^ 0xff, frame[index]);
   logWarning(buf);
 }
}

void generateErrorBCC1(unsigned char *frame){
 int errorFlag = (rand() % 100) < ERROR_PROBABILITY_BCC1;
 if (errorFlag)
 {
   char buf[MAX_SIZE];
   int index = (rand() % 2)+1;   //Index only in header field
   frame[index] = frame[index] ^ 0xff;         //Negates a byte
   sprintf(buf, "Generated BCC1 with errors.\n\t\t> Frame at index: %i = %#x =>
%#x\n", index, frame[index] ^ 0xff, frame[index]);
   logWarning(buf);
 }
}
```

utils.h

```
#ifndef UTILS_H
#define UTILS_H
```

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <signal.h>
#include <time.h>
#include "application.h"

#include "macros.h"

struct timeElapsed timeElapsed;

struct timeElapsed {
   clock_t start;
   clock_t end;
   double timeTaken;
};

int timeout,timeoutCount;
/**
* @brief Handle the Alarm; Checks whether timeout has been reached
* @return void
*/
void timeoutHandler();
/**
* @brief Disconects timeout .Sets timeout back to 1
* @return void
*/
void disconnectTimeout();
/**
* @brief Log Error.
*
* Prints error message in the right format
*
* @param msg    array to put the message
* @return void
*/
void logError(char *msg);
/**
* @brief Log Sucess.
*
```

```
* Prints Sucess message in the right format
*
* @param msg    array to put the message
* @return void
*/
void logSuccess(char *msg);
/**
* @brief Log Info.
*
* Prints message with information in the right format
*
* @param msg    array to put the message
* @return void
*/
void logInfo(char *msg);
/**
* @brief Log Warning.
*
* Prints Warning message in the right format
*
* @param msg    array to put the message
* @return void
*/
void logWarning(char *msg);
/**
* @brief Log Usage.
*
* Prints the correct usage of the application
*
* @return void
*/
void logUsage();

/**
* @brief Function that starts the timing (To calculate Bit Rate).
*
* @return void
*/
void startTimeElapsed();

/**
* @brief Function that ends the timing and calculates the Bit Rate.
*
* @return void
*/
```

```
void endTimeElapsed();



void logStats();
/**
* @brief Generate BCC2 Error, by changing one of the data bytes on Frame I, (must be used before transmitter sending the frame to receiver)
*
* @param frame      frame I that is ready to be sent
* @param size       size of frame
* @param stuffedBCC2Size  size of stuffed bbc2
* @return void
*/
void generateErrorBCC2(unsigned char *frame, int size, int stuffedBCC2Size);
/**
* @brief  Generate BCC1 Error, by changing either the A field or C Field on Frame I, (must be used before transmitter sending the frame to receiver)
*
* @param frame      frame I that is ready to be sent
* @return void
*/
void generateErrorBCC1(unsigned char *frame);

#endif
```

macros.h

```
#pragma once
#define MAX_SIZE 255
#define TRANSMITTER 0
#define RECEIVER    1
#define DATA_MAX_SIZE 1024
#define WORST_CASE_FRAME_I (DATA_MAX_SIZE*2 + 2 + 5) // Considering every data byte stuffed and BCC2 stuffed + 5 (Flag, field A, field C, BCC1, FLAG)
#define ERROR_PROBABILITY_BCC1  0;         //Error probability for BCC1 in percentage (range 0 to 100)
#define ERROR_PROBABILITY_BCC2  10;         //Error probability for BCC2 in percentage (range 0 to 100)

#define MAX_TIME 3               //  Tempo de espera até reenvio de trama SET pelo Emissor
#define MAX_TRIES 3
#define BAUDRATE B38400
#define TIME_OUT_SCS 3               //tempo maximo de espera para reenvio de trama SET por emissor
```

```
#define TIME_OUT_CHANCES 3        //Numero de tentativas de timeout


/*          Control Packge        */
#define CTRL_PACK_C_DATA    0x01
#define CTRL_PACK_C_START   0x02
#define CTRL_PACK_C_END     0x03

#define CTRL_PACK_T_SIZE    0x00
#define CTRL_PACK_T_NAME    0x01


/*      FLAG F    */
#define FLAG 0b01111110          //  (0x7E) Flag que marca inicio e fim de cada
Trama
// octeto A : endereco
#define A_CERR 0b00000011       // (0x03)Comandos enviados pelo Emissor e Respostas
enviadas pelo Receptor
#define A_CRRE 0b00000001       // (0x01)Comandos enviados pelo Receptor e
Respostas enviadas pelo Emissor

/*      Campo de controlo C     */

//Tramas de Informacao
#define C_I(s) ((s == 0)? 0x00 : 0x40)    // s = numero de sequencia em tramas I

//Tramas de comando
#define C_SET 0b00000011                    // (0x03)setup
#define C_DISC 0b00001011                   // (0x0B)disconnect
//Tramas de resposta
#define C_UA 0b00000111                     // (0x07)unnumbered acknowledgment
#define C_RR(r) ((r == 0)? 0x05 : 0x85)     // (0x05 or 0x85)receiver ready /
positive ACK
#define C_REJ(r) ((r == 0)? 0x01 : 0x81)    // (0x81 or 0x01)reject / negative ACK

/*      Campo de Proteção (cabeçalho)    */

#define BCC(a,c) (a ^ c)                    // XOR entre Campo A e C
#define CS(seq) ((seq == 0)? 0x0 : 0x40)


/*              Identity                    */
#define SENDER 0
#define RECEIVER 1
```

```
/*      Stuffed     */
#define ESCAPE 0x7d
#define FLAG_ESC 0x5E
#define ESC_ESC 0x5D

/*      Errors          */
#define INCORRECT_C_FIELD -1
#define INCORRECT_BCC1_FIELD -2
```