

**Redes de Computadores**

**1º Trabalho Laboratorial – Ligação de dados**

**Mestrado Integrado em Engenharia Informática e computação**

**10 de Novembro de 2020**

**Diogo Guimarães do Rosário - up201806582**

**Henrique Melo Ribeiro - up201806529**

**Índice**

**Sumário** 3

**Introdução** 3

**Arquitetura e Estrutura de código** 4

Camada do protocolo

Camada de aplicação

**Casos de uso principais 5**

**Protocolo de ligação lógica 5**

Configuração da porta série

Estabelecer a conexão entre as duas portas série

Transferência dos pacotes de dados após operações de stuffing e destuffing

Deteção de erros nas transmissões

**Protocolo de aplicação 7**

Criação pacotes de controlo e de informação a partir da leitura do ficheiro

Leitura e escrita do ficheiro

Controlo do programa através das funções llread(), llwrite() ,llopen() e llclose();

**Validação 8**

**Eficiência do protocolo de ligação de dados 8**

Variação do FER

Variação do tamanho das tramas

**Conclusão 10**

**Sumário**

Este trabalho foi desenvolvido no âmbito da unidade curricular Redes de computadores.

O projeto consistia no desenvolvimento de uma aplicação capaz de transferir dados de um computador para outro através de uma porta série assíncrona. A aplicação é resistente a erros na transmissão dos pacotes de dados e desconexão da porta série.

A aplicação foi desenvolvida com sucesso, sendo possível transferir ficheiros entre dois computadores sem qualquer perda de informação.

**1. Introdução**

Este relatório tem o propósito de expor o modo como a nossa aplicação está organizada bem como o funcionamento desta.

O objetivo deste trabalho é implementar um protocolo de ligação de dados especificado no guião do trabalho, de modo a permitir transferência fiável de dados entre dois dispositivos conectados pela porta série.

Assim o relatório estará organizado da seguinte forma:

2. Arquitetura e Estrutura do código - Demonstração dos blocos funcionais e interfaces e exposição das principais estruturas de dados, funções e sua relação com a arquitetura

3. Casos de uso principais - Identificação das sequências de chamada de funções

4. Protocolo de ligação lógica - Identificação dos principais aspetos funcionais, descrição da estratégia de implementação destes aspetos com apresentação de extratos de código

5. Protocolo de aplicação - Identificação dos principais aspetos funcionais, descrição da estratégia de implementação destes aspetos com apresentação de extratos de código

6. Validação - Descrição dos testes efetuados com apresentação dos resultados

7. Eficiência do protocolo de ligação de dados - Caraterização estatística da eficiência do protocolo

8. Conclusões - Síntese da informação apresentada nas secções anteriores

**2. Arquitetura e Estrutura de código**

O nosso projeto foi desenvolvido com duas camadas principais (protocolo e aplicação).

**2.1. Camada do protocolo**

A camado do protocolo está definida no ficheiro common.h e é a camada de nível mais baixo do nosso programa. É responsável pela comunicação entre os dois computadores através da porta série.

Para além disso, faz uso dos ficheiros writenoncanonical.c, writenoncanonical.h, noncanonical.c e noncanonical.h.

Esta separação foi feita devido a certas funções não serem necessárias dos dois lados do protocolo (escrita e leitura), como por exemplo a função de leitura de dados e a sua máquina de estados.

Foi utilizada a seguinte estrutura de dados para facilitar este processo:

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\**/

  int currentTry;                /*\*Número da tentativa atual em caso de falha\**/

  char frame[MAX\_SIZE\*2+7];          /*\*Trama\**/

  int frameSize;

};

**2.2. Camada de aplicação**

A camada da aplicação está definida nos ficheiros application.c e application.h e é a camada que está imediatamente acima do protocolo.

Esta camada é responsável pela leitura do ficheiro no lado da escrita e criar os pacotes de dados que serão transmitidos à camada do protocolo, do lado de leitura é responsável por receber os pacotes da camada de protocolo e escreve-los para um ficheiro.

Foi utilizada a seguinte estrutura de dados para facilitar este processo:

struct applicationLayer

{

    int fileDescriptor; /\*Descritor correspondente à porta série\*/

    int status;         /\*TRANSMITTER | RECEIVER\*/

};

**3. Casos de uso principais**

A aplicação necessita de diferentes parâmetros dependendo se é recetor ou transmissor, devendo a ordem destes ser respeitada.

Do lado do transmissor é necessário o número da porta série a ser usada, a string "transmitter" e o nome do ficheiro.

Exemplo : ./application 0 transmitter pinguim.gif

Do lado do recetor é necessário o número da porta série a ser usada e a string "receiver".

Exemplo : ./application 0 receiver

O recetor quando iniciado fica à espera de um transmissor para iniciar a conexão. Após estabelecer esta conexão o ficheiro começa a ser transmitido pelo transmissor em pacotes de dados.

**4. Protocolo de ligação lógica**

O protocolo de ligação lógica tem como objetivos:

\* Configurar a porta série;

\* Estabelecer a conexão entre as duas portas série;

\* Transferência dos pacotes de dados após operações de *stuffing* e *destuffing;*

\* Deteção de erros nas transmissões;

**4.1. Configuração da porta série**

**openWriter() e openReader()**

Estas funções recebem como argumento a porta séria com a qual será estabelecida uma ligação e são invocadas na função **llopen()** sendo que esta função consegue distinguir qual o estado do programa (transmitter ou receiver) que a chama.

Pala além do referido anteriormente também são responsáveis por preencher a struct da camada de protocolo com os valores corretos.

Caso seja impossível alcançar a porta série desejada o programa lança uma mensagem de erro e exit code -1.

**4.2. Estabelecer a conexão entre as duas portas série**

**setupWriterConnection() e setupReaderConnection()**

Estas funções são também invocadas pelo **llopen()** que lhes fornece o descritor das portas abertas pelas funções **openWriter()** ou **openReader()**.

O estabelecimento da conexão começa com o programa emissor a enviar uma mensagem de controlo *set* que quando recebida e processada pelo programa recetor responde com uma mensagem *UA.* Estas mensagens são processadas nas states machines correspondentes e enviadas pela função sendSupervisionPacket().

Nestas funções também se dá *setup* ao sinal de alarm (SIG\_ALARM). O alarm serve como controlo nas comunicações, fazendo o emissor reenviar a última mensagem escrita caso este não receba resposta por parte do recetor. Existe uma ocasião em que os papéis estão invertidos, sendo esta quando os programas se estão a desconectar da porta série.

**4.3. Transferência dos pacotes de dados após operações de *stuffing* e *destuffing***

Esta transferência é garantida pelas função **sendInfo()** e **readInfo()**.

A função **sendInfo()** da camada de protocolo recebe o descritor da porta série, a mensagem e o seu tamanho, e retorna o número de bytes escritos. Nesta função são feitas operações de *stuffing* da mensagem de forma a garantir a comunicação correta entre programas.

Depois de escrever para a porta série a função **readRR()** é invocada, fazendo o emissor esperar por uma resposta enviada pelo recetor. Se esta mensagem recebida for uma mensagem *receiver ready* o programa passa para o tratamento do próximo conjunto de dados que tem de enviar, caso seja de *reject* o programa reenvia a última mensagem e volta a ficar à espera de uma mensagem.

A função **readInfo()** responsabiliza-se por fazer a leitura da informação da porta série e de enviar a resposta correta ao programa emissor. Na função **readInfo()** são feitas operações de *destuffing* de forma a restaurar a mensagem recebida para o estado anterior ao *stuffing*  que esta sofreu.

**4.4. Deteção de erros nas transmissões**

A deteção de erros é feita ao mesmo tempo que a leitura da mensagem é feita, comparando a mensagem recebida com o **BCC** final (o **BCC** é igual à operação lógica ‘ou exclusivo’ entre todos os bytes da informação), e verificando o **número de série** da mensagem (varia entre 0 e 1 quando a mensagem recebida conter a informação correta).

No caso em que **BCC** **da informação** (BCC2) está errado é enviada uma resposta *reject.* Caso o **número de** **série** esteja errado a informação recebida é considerada repetida e é enviada uma resposta *receiver ready* para o emissor enviar o próximo pacote de informação. No caso em que o **BCC do cabeçalho** (BCC1) falha, a trama é ignorada esperando que o emissor reenvie a mensagem.

**5. Protocolo de aplicação**

O protocolo de aplicação tem como objetivos:

\* Gerar pacotes de controlo e de informação a partir da leitura do ficheiro;

\* Leitura e escrita do ficheiro;

\* Controlo do programa através das funções llread(), llwrite() ,llopen() e llclose();

**5.1. Criação pacotes de controlo e de informação a partir da leitura do ficheiro**

A criação de pacotes de controlo é realizada na função **buildControlPacket(),** função essa que recebe como argumentos o nome do ficheiro e o seu tamanho, bem como o byte do controlo (CONTROL\_START E CONTROL\_END no ficheiro application.h) e um buffer a preencher que corresponderá ao pacote completo. Este pacote será codificado no formato TLV (Type, length, value).

A criação de pacotes de informação é realizada na função **buildDataPacket(),** que receberá como parâmetros um *chunk* de tamanho MAX\_SIZE – 4 de modo a reservar 4 bytes para o byte de controlo, número de série e tamanho de bytes de informação. Este tamanho MAX\_SIZE corresponde ao tamanho que será lido de cada vez no ficheiro e está definido no ficheiro common.h.

**5.2. Leitura e escrita do ficheiro**

A leitura do ficheiro só é realizada nos casos em que a aplicação é inicializada com o valor transmitter e é realizada num loop até o ficheiro ser inteiramente lido. Posteriormente a cada iteração do loop, é construído um pacote com esta informação e enviada para o protocolo de dados através da função **llwrite().** Do lado do recetor, a informação é lida e guardada num array com o tamanho do ficheiro alocado aquando da leitura do primeiro pacote de controlo, visto que neste pacote é recebida a informação do nome e tamanho do ficheiro.

No final do programa este array é inteiramente escrito para o ficheiro usando **fwrite(),** criando assim o ficheiro com toda a informação recebida.

**5.3. Controlo do programa através das funções llread(), llwrite(), llopen() e llclose();**

Inicialmente, ambos os programas invocam **llopen()**, de modo a abrir a porta série e iniciar a comunicação destas através das sequências *set* e *UA*. Posteriormente é feito um loop na qual o transmissor lê o ficheiro e invoca **llwrite()**, enquanto que o recetor apenas invoca **llread()** também num loop controlado pela flag finished. Aquando da leitura do pacote de controlo final, verifica-se que este pacote é igual ao pacote controlo inicial e caso seja, a flag finished será posta a true, acabando o loop de leitura.

Posteriormente, é invocado **llclose()** nos 2 lados da aplicação, desconectando a porta série e escrevendo o ficheiro recebido do lado do recetor.

**6. Validação.**

Para verificar a integridade do nosso código, o programa foi sujeito a diversos testes:

-Passagem de um ficheiro com extensão .gif (11 KB)

-Passagem de um ficheiro com extensão .mp4 (13 MB)

-Interrupção do programa através de time-out.

-Desligar a o cabo da porta série aquando da transferência de um ficheiro sem a ligar novamente.

-Desligar a o cabo da porta série aquando da transferência de um ficheiro, voltando a ligá-lo novamente.

-Geração de erros na ligação da porta série.

-Variação do tamanho máximo da trama de informação (testes realizados com 250, 500 e 1000 bytes).

-Variação do baud rate da porta série.

Perante todos estes testes o programa conseguiu terminar com sucesso sendo posteriormente verificada esta integridade usando o comando *diff* nativo do Linux, estando os ficheiros idênticos.

**7. Eficiência do protocolo de ligação de dados**

**7.1. Variação do FER**

Como se pode observar no seguinte gráfico, os erros causados no BCC1 e BCC2 (erros causados usando valores aleatórios) têm um impacto grande na eficiência do programa.

Após uma análise mais detalhada é possível observar que os erros que causam maior impacto na eficiência são os erros causados no BCC1 (cada erro causa por volta de 1 segundo de atraso visto que o programa terá de esperar pelo sinal de alarme para reenviar a informação) enquanto os erros causados pelo BCC2 quase que não têm impacto na eficiência do programa, visto que quando isto acontece, o recetor envia uma mensagem de *reject* que faz com que o emissor reenvie a mensagem imediatamente.

Gráfico obtido usando um ficheiro de tamanho 10968 bytes, baudrate 38400 e chunk size de 250. O eixo x corresponde à taxa de erros causados no BCC1 (erros no BCC do cabeçalho) e BCC2 (erros no BCC de informação) e o y ao FER.

**7.2. Variação do tamanho das tramas**

Como se pode verificar pelo gráfico a baixo, ao aumentar o tamanho da trama, a eficiência do programa aumenta, até atingir um valor em que, para se notar melhoramentos significativos, seria necessário alterar outros valores.

No eixo x estão representados os vários tamanhos das tramas e no eixo y estão representadas as várias eficiências.

**7.3. Variação do baud rate**

Como se pode verificar pelo gráfico a baixo, o baud rate tem uma influência insignificativa na eficiência do programa, apesar de o tempo ser inversamente proporcional ao baud rate.

No eixo x estão representados os vários baud rates e no eixo y a eficiência.

**8. Conclusão**

O fundamento deste trabalho é a passagem de informação entre processos garantindo a sua integridade.

Algo muito importante para este fim é a separação de tarefas entre a camada de aplicação que trata da leitura e escrita de ficheiros e da camada de protocolo que trata da comunicação entre processos através de uma porta série, sem que nenhuma precise de saber os detalhes da outra, conseguindo realizar a sua funcionalidade com os dados fornecidos.

Em suma, pensamos que o objetivo global deste trabalho foi alcançado. Apesar de no início do projeto sentirmos que os fundamentos necessários para realizar o projeto fossem muito complexos após algumas leituras do enunciado e trocas de ideias com colegas achamos que conseguimos entender bem o necessário e colocar este conhecimento em prática.

***Anexo I***

Application.h

#pragma once

#define TRANSMITTER 0

#define RECEIVER 1

#define DATA\_C 0x01

#define CONTROL\_START 0x02

#define CONTROL\_END 0x03

#define FILESIZE 0

#define FILENAME 1

#define READ\_C 0

#define READ\_NEXT\_START 1

#define READ\_NEXT\_END 2

#define READ\_DATA 3

struct applicationLayer

{

    int fileDescriptor; /\*Descritor correspondente à porta série\*/

    int status;         /\*TRANSMITTER | RECEIVER\*/

};

int llopen(char \* port, int status);

int llwrite(int fd, unsigned char \* buffer,int length);

int llread(int fd, unsigned char \* buffer);

int buildDataPacket(char \*buf, unsigned char \*packet, int length);

int buildControlPacket(char \*filename, long filesize, unsigned char \*pack, char controlField);

int decodeAppPacket(unsigned char \*appPacket, int bytesRead);

int llclose(int fd);

Application.c

#include "writenoncanonical.h"

#include "noncanonical.h"

#include "application.h"

#include "common.h"

struct applicationLayer app;

int serialNumber = 0;

char \*fileName; //saves the file name to compare at the end

long fileSize; //saves the file size to compare at the end

char \*writeToFile;

int currentFileArrayIndex;

int finished = FALSE; //indicates if the 'end' packed has been received

int llopen(char \*port, int status)

{

    int fd;

    if (status == TRANSMITTER)

    {

        fd = openWriter(port);

        setupWriterConnection(fd);

    }

    if (status == RECEIVER)

    {

        fd = openReader(port);

        setupReaderConnection(fd);

    }

    return fd;

}

int llwrite(int fd, unsigned char \*buffer, int length)

{

    int bytesRead = sendInfo(buffer, length, fd);

    readRR(app.fileDescriptor);

    return bytesRead;

}

int llread(int fd, unsigned char \*appPacket)

{

    int size = readInfo(app.fileDescriptor, appPacket);

    return size;

}

int buildDataPacket(char \*buf, unsigned char \*packet, int length)

{

    packet[0] = DATA\_C;

    packet[1] = serialNumber % 255;

    packet[2] = length / 256;

    packet[3] = length % 256;

    int i = 0;

    for (; i < length; i++)

    {

        packet[4 + i] = buf[i];

    }

    serialNumber++;

    return 4 + i;

}

int buildControlPacket(char \*filename, long filesize, unsigned char \*pack, char controlField)

{

    pack[0] = controlField;

    pack[1] = FILESIZE;

    pack[2] = sizeof(long);

    memcpy(pack + 3, &filesize, sizeof(long));

    pack[3 + sizeof(long)] = FILENAME;

    pack[4 + sizeof(long)] = strlen(filename);

    memcpy(pack + 5 + sizeof(long), filename, strlen(filename));

    int sizeWritten = 5 + sizeof(long) + strlen(filename);

    return sizeWritten;

}

int decodeAppPacket(unsigned char \*appPacket, int bytesRead)

{

    int state = READ\_C;

    static int nextPackSequenceNumber = 0;

    int bytesToSkip = 0;

    int filenameOK = FALSE;

    int filesizeOK = FALSE;

    while (1 + bytesToSkip < bytesRead)

    {

        switch (state)

        {

        case READ\_C:

            if (appPacket[0] == CONTROL\_START)

            {

                state = READ\_NEXT\_START;

            }

            else if (appPacket[0] == CONTROL\_END)

            {

                state = READ\_NEXT\_END;

            }

            else if (appPacket[0] == DATA\_C)

            {

                state = READ\_DATA;

            }

            else

            {

                bytesToSkip = 100000;

            }

            break;

        case READ\_NEXT\_START:

            if (appPacket[1 + bytesToSkip] == FILENAME)

            {

                int size = (int)appPacket[2 + bytesToSkip];

                fileName = malloc(sizeof(char) \* size);

                for (int j = 0; j < size; j++)

                {

                    fileName[j] = appPacket[3 + bytesToSkip + j];

                }

                bytesToSkip += (2 + size);

            }

            else if (appPacket[1 + bytesToSkip] == FILESIZE)

            {

                int size = (int)appPacket[2 + bytesToSkip];

                unsigned char \*filesize = malloc(sizeof(char) \* size);

                for (int i = 0; i < size; i++)

                {

                    filesize[i] = appPacket[3 + bytesToSkip + i];

                    fileSize |= (filesize[i] << 8 \* i);

                }

                writeToFile = malloc(sizeof(char) \* fileSize);

                bytesToSkip += (2 + size);

            }

            break;

        case READ\_NEXT\_END:

            if (appPacket[1 + bytesToSkip] == FILENAME)

            {

                int size = (int)appPacket[2 + bytesToSkip];

                char \*filename = malloc(sizeof(char) \* size);

                for (int i = 0; i < size; i++)

                {

                    filename[i] = appPacket[3 + bytesToSkip + i];

                }

                if (!strcmp(filename, fileName))

                {

                    filenameOK = TRUE;

                }

                free(filename);

                bytesToSkip += (2 + size);

                if (bytesToSkip + 1 >= bytesRead)

                {

                    finished = TRUE;

                }

            }

            else if (appPacket[1 + bytesToSkip] == FILESIZE)

            {

                int size = (int)appPacket[2 + bytesToSkip];

                unsigned char \*filesize = malloc(sizeof(char) \* size);

                long aux = 0;

                for (int i = 0; i < size; i++)

                {

                    filesize[i] = appPacket[3 + bytesToSkip + i];

                    aux |= (filesize[i] << 8 \* i);

                }

                if (aux == fileSize)

                {

                    filesizeOK = TRUE;

                }

                bytesToSkip += (2 + size);

                if (bytesToSkip + 1 >= bytesRead)

                {

                    finished = TRUE;

                }

            }

            break;

        case READ\_DATA:

            if ((int)appPacket[1] == nextPackSequenceNumber)

            {

                int size = (int)appPacket[2] \* 256;

                size += (int)appPacket[3];

                bytesToSkip += 3 + size;

                for (int i = 0; i < size; i++)

                {

                    writeToFile[i + currentFileArrayIndex] = appPacket[4 + i];

                }

                currentFileArrayIndex += size;

                nextPackSequenceNumber = (nextPackSequenceNumber + 1) % 255;

            }

            else

            {

                bytesToSkip = 10000;

            }

            break;

        }

    }

    if (filesizeOK && filenameOK)

    {

        printf("Control Packets matched, file received\n");

    }

    return TRUE;

}

int llclose(int fd)

{

    if (app.status == TRANSMITTER)

    {

        return transmitterDisconnect(fd);

    }

    if (app.status == RECEIVER)

    {

        return receiverDisconnect(fd);

    }

    return -1;

}

int main(int argc, char \*\*argv)

{

    FILE \*f1;

    if (!strcmp(argv[2], "transmitter"))

    {

        if (argc < 4)

        {

            printf("Usage : ./application [port] transmitter [path\_to\_file]");

            exit(1);

        }

        f1 = fopen(argv[3], "r");

        if(f1 == NULL){

            printf("File does not exist\n");

            exit(1);

        }

        app.status = TRANSMITTER;

        char port[20] = "/dev/ttyS";

        char portNumber[20];

        strcpy(portNumber,argv[1]);

        strcat(port,portNumber);

        app.fileDescriptor = llopen(port, TRANSMITTER);

    }

    else if (!strcmp(argv[2], "receiver"))

    {

        if (argc < 2)

        {

            printf("Usage : ./application [port] receiver");

            exit(1);

        }

        app.status = RECEIVER;

        char port[20] = "/dev/ttyS";

        char portNumber[20];

        strcpy(portNumber,argv[1]);

        strcat(port,portNumber);

        app.fileDescriptor = llopen(port, RECEIVER);

    }

    else

    {

        printf("Usage : ./application [port] transmitter [path\_to\_file] or ./application [port] receiver");

        exit(1);

    }

    if (!strcmp(argv[2], "transmitter"))

    {

        unsigned char pack[MAX\_SIZE];

        fseek(f1, 0, SEEK\_END);

        long filesize = ftell(f1);

        int packSize = buildControlPacket(argv[3], filesize, pack, CONTROL\_START);

        printf("Started sending file\n");

        llwrite(app.fileDescriptor, pack, packSize);

        fseek(f1, 0, SEEK\_SET);

        int sizeRemaining = filesize;

        while (sizeRemaining > 0)

        {

            unsigned char packet[MAX\_SIZE];

            char buf[MAX\_SIZE - 4];

            int bytesRead = fread(buf, 1, MAX\_SIZE - 4, f1);

            int size = buildDataPacket(buf, packet, bytesRead);

            sizeRemaining -= bytesRead;

            llwrite(app.fileDescriptor, packet, size);

        }

        packSize = buildControlPacket(argv[3], filesize, pack, CONTROL\_END);

        llwrite(app.fileDescriptor, pack, packSize);

        printf("Finished sending file\n");

        llclose(app.fileDescriptor);

        closeWriter(app.fileDescriptor);

    }

    else if (!strcmp(argv[2], "receiver"))

    {

        printf("Started receiving file\n");

        while (!finished)

        {

            unsigned char appPacket[MAX\_SIZE];

            int bytesRead = llread(app.fileDescriptor, appPacket);

            decodeAppPacket(appPacket, bytesRead);

        }

        FILE \*newFile;

        char path[256] = "../";

        strcat(path, fileName);

        newFile = fopen(path, "wb");

        fwrite(writeToFile, sizeof(char), fileSize, newFile);

        fclose(newFile);

        printf("File saved in %s\n",path);

        llclose(app.fileDescriptor);

        closeReader(app.fileDescriptor);

        free(writeToFile);

        free(fileName);

    }

}

writenoncanonical.h

##pragma once

int sendInfo(unsigned char \*info, int size, int fd);

int stuffChar(char info, unsigned char \*buf);

int setupWriterConnection(int fd);

int openWriter(char \* port);

void closeWriter(int fd);

int readRR(int fd);

int readUA(int fd);

int transmitterDisconnect(int fd);

int writerReadDISC(int status, int fd);

int rrStateMachine(unsigned char \*buf, int fd);

writenoncanonical.c

/\*Non-Canonical Input Processing\*/

#include "writenoncanonical.h"

#include <sys/types.h>

#include "common.h"

#include "application.h"

struct linkLayer protocol;

struct termios oldtio, newtio;

static int currentState = 0; //current state in the state machine

static int currentIndex = 0; //index of the size of the packet received

static int activatedAlarm = FALSE; //indicates if the alarm has been triggered

static volatile int STOP = FALSE;

static void atende(int signo) // atende alarme

{

  switch (signo){

    case SIGALRM:

      if(protocol.currentTry>=protocol.numTransmissions)

        exit(1);

      protocol.currentTry++;

      activatedAlarm = TRUE;

      break;

  }

}

int writerReadDISC(int status, int fd)

{

  STOP=FALSE;

  unsigned char recvBuf[5];

  while (STOP == FALSE)

  {

    read(fd, recvBuf, 1);

    if(activatedAlarm)

    {

      activatedAlarm = FALSE;

      write(fd,protocol.frame,protocol.frameSize);

      alarm(protocol.timeout);

    }

    if (discStateMachine(status, recvBuf, &currentState,&currentIndex))

    {

      if (currentState == DONE)

      {

        alarm(0);

        protocol.currentTry = 0;

        currentIndex = 0;

        currentState = START;

        STOP = TRUE;

      }

    }

  }

  return 0;

}

int openWriter(char \* port)

{

    fillProtocol(&protocol, port, 0);

    int fd = open(protocol.port, O\_RDWR | O\_NOCTTY);

    if (fd < 0)

    {

        perror(protocol.port);

        exit(-1);

    }

    if (tcgetattr(fd, &oldtio) == -1)

    { /\* save current port settings \*/

        perror("tcgetattr");

        exit(-1);

    }

    bzero(&newtio, sizeof(newtio));

    newtio.c\_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;

    newtio.c\_iflag = IGNPAR;

    newtio.c\_oflag = 0;

    /\* set input mode (non-canonical, no echo,...) \*/

    newtio.c\_lflag = 0;

    newtio.c\_cc[VTIME] = 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)

    {

        perror("tcsetattr");

        exit(-1);

    }

    printf("Connection established at: %s\n",port);

    return fd;

}

void closeWriter(int fd)

{

   sleep(1);

    if (tcsetattr(fd, TCSANOW, &oldtio) == -1)

    {

        perror("tcsetattr");

        exit(-1);

    }

}

int stuffChar(char info, unsigned char \*buf)

{

  if (info == FLAG)

  {

    buf[0] = ESCAPE;

    buf[1] = ESCAPEFLAG;

    return TRUE;

  }

  else if (info == ESCAPE)

  {

    buf[0] = ESCAPE;

    buf[1] = ESCAPEESCAPE;

    return TRUE;

  }

  else

  {

    buf[0] = info;

    return FALSE;

  }

}

int sendInfo(unsigned char \*info, int size, int fd)

{

  int res = 0;

  int currentPos = 4;

  char sendMessage[size\*2+7];

  sendMessage[0] = FLAG;

  sendMessage[1] = SENDER\_A;

  if (protocol.sequenceNumber == 1)

    sendMessage[2] = INFO\_C\_1;

  else if (protocol.sequenceNumber == 0)

    sendMessage[2] = INFO\_C\_0;

  sendMessage[3] = SENDER\_A ^ sendMessage[2];

  unsigned char bcc = '\0';

  int offset = 0;

  for (int i = 0; i < size; i++)

  {

    bcc = bcc ^ info[i];

    unsigned char stuffedBuf[2];

    if (stuffChar(info[i], stuffedBuf))

    {

      sendMessage[4 + i + offset] = stuffedBuf[0];

      offset++;

      sendMessage[4 + i + offset] = stuffedBuf[1];

    }

    else

      sendMessage[4 + i + offset] = stuffedBuf[0];

    currentPos++;

  }

  unsigned char stuffedBCC[2];

  if (stuffChar(bcc, stuffedBCC))

  {

    sendMessage[currentPos + offset] = stuffedBCC[0];

    offset++;

    sendMessage[currentPos + offset] = stuffedBCC[1];

  }

  else

    sendMessage[currentPos + offset] = stuffedBCC[0];

  currentPos++;

  sendMessage[currentPos + offset] = FLAG;

  currentPos++;

  res = write(fd, sendMessage, currentPos + offset + 1);

  protocol.frame[0] = '\0';

  memcpy(protocol.frame, sendMessage, currentPos + offset + 1);

  protocol.frameSize = currentPos + offset + 1;

  alarm(protocol.timeout);

  return res;

}

int transmitterDisconnect(int fd)

{

  sendSupervisionPacket(SENDER\_A, DISC\_C, &protocol,fd);

  alarm(protocol.timeout);

  protocol.currentTry = 0;

  writerReadDISC(RECEIVER\_A,fd);

  sendSupervisionPacket(RECEIVER\_A,UA\_C, &protocol,fd);

  printf("Connection closed\n");

  return 1;

}

int rrStateMachine(unsigned char \*buf, int fd)

{

  switch (currentState)

  {

  case START:

    if (buf[0] == FLAG)

    {

      currentIndex++;

      currentState = FLAG\_RCV;

      return TRUE;

    }

    break;

  case FLAG\_RCV:

    if (buf[0] == SENDER\_A)

    {

      currentIndex++;

      currentState = A\_RCV;

      return TRUE;

    }

    else if (buf[0] == FLAG)

      return TRUE;

    else

    {

      currentIndex = 0;

      currentState = START;

    }

    break;

  case A\_RCV:

    if (protocol.sequenceNumber == 0 && buf[0] == RR\_C\_1)

    {

      currentIndex++;

      currentState = C\_RCV;

      return TRUE;

    }

    else if (protocol.sequenceNumber == 1 && buf[0] == RR\_C\_0)

    {

      currentIndex++;

      currentState = C\_RCV;

      return TRUE;

    }

    else if (protocol.sequenceNumber == 0 && buf[0] == REJ\_C\_1)

    {

      protocol.currentTry=0;

      write(fd,protocol.frame,protocol.frameSize);

      alarm(protocol.timeout);

      currentState = START;

      currentIndex = 0;

      return FALSE;

    }

    else if (protocol.sequenceNumber == 1 && buf[0] == REJ\_C\_0)

    {

      write(fd,protocol.frame,protocol.frameSize);

      alarm(protocol.timeout);

      protocol.currentTry=0;

      currentState = START;

      currentIndex = 0;

      return FALSE;

    }

    else if (buf[0] == FLAG)

    {

      currentIndex = 0;

      currentState = FLAG\_RCV;

      return TRUE;

    }

    else

    {

      currentIndex = 0;

      currentState = START;

    }

    break;

  case C\_RCV:

    if (protocol.sequenceNumber == 0 && buf[0] == (RR\_C\_1 ^ SENDER\_A))

    {

      currentIndex++;

      currentState = BCC\_OK;

      return TRUE;

    }

    else if (protocol.sequenceNumber == 1 && buf[0] == (RR\_C\_0 ^ SENDER\_A))

    {

      currentIndex++;

      currentState = BCC\_OK;

      return TRUE;

    }

    else if (buf[0] == FLAG)

    {

      currentIndex = 0;

      currentState = FLAG\_RCV;

      return TRUE;

    }

    else

    {

      currentIndex = 0;

      currentState = START;

    }

    break;

  case BCC\_OK:

    if (buf[0] == FLAG)

    {

      currentIndex++;

      currentState = DONE;

      return TRUE;

    }

    else

    {

      currentIndex = 0;

      currentState = START;

    }

    break;

  default:

    break;

  }

  currentIndex = 0;

  return FALSE;

}

int writerReadUA(int status,int fd)

{

  STOP=FALSE;

  unsigned char recvBuf[5];

  while (STOP == FALSE)

  {

    read(fd, recvBuf, 1);

    if(activatedAlarm)

    {

      activatedAlarm = FALSE;

      write(fd,protocol.frame,protocol.frameSize);

      alarm(protocol.timeout);

    }

    if (UAStateMachine(recvBuf,status,&currentState,&currentIndex))

    {

      if (currentState == DONE)

      {

        alarm(0);

        protocol.currentTry = 0;

        currentIndex = 0;

        currentState = START;

        STOP = TRUE;

      }

    }

  }

  return 0;

}

int readRR(int fd)

{

  unsigned char recvBuf[5];

  STOP = FALSE;

  while (STOP == FALSE)

  {

    read(fd, recvBuf, 1);

    if(activatedAlarm)

    {

      activatedAlarm = FALSE;

      write(fd,protocol.frame,protocol.frameSize);

      alarm(protocol.timeout);

    }

    if (rrStateMachine(recvBuf,fd))

    {

      if (currentState == DONE)

      {

        alarm(0);

        protocol.currentTry = 0;

        updateSequenceNumber(&protocol);

        currentIndex = 0;

        currentState = START;

        STOP = TRUE;

      }

    }

  }

  return 0;

}

int setupWriterConnection(int fd)

{

  sendSupervisionPacket(SENDER\_A, SET\_C, &protocol, fd);

  struct sigaction psa;

  psa.sa\_handler = atende;

  sigemptyset(&psa.sa\_mask);

  psa.sa\_flags=0;

  sigaction(SIGALRM, &psa, NULL);

  alarm(protocol.timeout);

  writerReadUA(SENDER\_A,fd);

  return 0;

}

noncanonical.h

#pragma once

int verifyBCC();

int readInfo(int fd, unsigned char \* appPacket);

int readSET();

void setupReaderConnection(int fd);

int openReader(char \* port);

void closeReader(int fd);

int sendReaderDISC(int fd);

int receiverDisconnect(int fd);

int infoStateMachine(unsigned char \*buf, int fd, unsigned char \* msg);

int receiverReadUA(int status, int fd);

int setStateMachine(char \*buf);

noncanonical.c

/\*Non-Canonical Input Processing\*/

#include "noncanonical.h"

#include <sys/types.h>

#include "common.h"

struct linkLayer protocol;

struct termios oldtio, newtio;

static int currentState = 0;

static int currentIndex = 0;

static volatile int STOP = FALSE;

int activatedAlarm = FALSE;

static void atende(int signo) // atende alarme

{

  switch (signo)

  {

  case SIGALRM:

    if (protocol.currentTry >= protocol.numTransmissions)

      exit(1);

    protocol.currentTry++;

    activatedAlarm = TRUE;

    break;

  }

}

int verifyBCC(unsigned char\* msg)

{

  unsigned char bccControl = '\0';

  for (int i = 0; i < currentIndex - 1; i++)

  {

    bccControl ^= msg[i];

  }

  if (msg[currentIndex - 1] == bccControl)

  {

    return TRUE;

  }

  return FALSE;

}

int infoStateMachine(unsigned char \*buf, int fd, unsigned char \* msg)

{

  static char C;

  switch (currentState)

  {

  case START:

    if (buf[0] == FLAG)

    {

      C = '\0';

      currentIndex = 0;

      currentState = FLAG\_RCV;

      return TRUE;

    }

    break;

  case FLAG\_RCV:

    if (buf[0] == SENDER\_A)

    {

      currentState = A\_RCV;

      return TRUE;

    }

    else if (buf[0] == FLAG)

      return TRUE;

    else

    {

      currentState = START;

    }

    break;

  case A\_RCV:

    if (protocol.sequenceNumber == 1 && buf[0] == INFO\_C\_0)

    {

      C = INFO\_C\_0;

      currentState = C\_RCV;

      return TRUE;

    }

    else if (protocol.sequenceNumber == 0 && buf[0] == INFO\_C\_1)

    {

      C = INFO\_C\_1;

      currentState = C\_RCV;

      return TRUE;

    }

    else if ((protocol.sequenceNumber == 0 && buf[0] == INFO\_C\_0) || (protocol.sequenceNumber == 1 && buf[0] == INFO\_C\_1))

    {

      C = '\0';

      currentState = START;

      if (protocol.sequenceNumber == 1)

        sendSupervisionPacket(SENDER\_A, RR\_C\_0, &protocol, fd);

      else if (protocol.sequenceNumber == 0)

        sendSupervisionPacket(SENDER\_A, RR\_C\_1, &protocol, fd);

      return TRUE;

    }

    else if (buf[0] == FLAG)

    {

      currentState = FLAG\_RCV;

      return TRUE;

    }

    else

    {

      currentState = START;

    }

    break;

  case C\_RCV:

    if (protocol.sequenceNumber == 1 && buf[0] == (SENDER\_A ^ INFO\_C\_0))

    {

      currentState = BCC\_OK;

      return TRUE;

    }

    else if (protocol.sequenceNumber == 0 && buf[0] == (SENDER\_A ^ INFO\_C\_1))

    {

      currentState = BCC\_OK;

      return TRUE;

    }

    else if (buf[0] == FLAG)

    {

      currentState = FLAG\_RCV;

      return TRUE;

    }

    else

    {

      currentState = START;

    }

  case BCC\_OK: //receives info

    if (buf[0] == FLAG)

    {

      if (verifyBCC(msg))

      {

        currentState = DONE;

        if (protocol.sequenceNumber == 1 && C == INFO\_C\_1)

        {

          msg[0] = '\0';

        }

        else if (protocol.sequenceNumber == 0 && C == INFO\_C\_0)

        {

          msg[0] = '\0';

        }

        else

        {

          if (protocol.sequenceNumber == 0)

            sendSupervisionPacket(SENDER\_A, RR\_C\_0, &protocol, fd);

          else if (protocol.sequenceNumber == 1)

            sendSupervisionPacket(SENDER\_A, RR\_C\_1, &protocol, fd);

          updateSequenceNumber(&protocol);

        }

        return TRUE;

      }

      else

      {

        if (protocol.sequenceNumber == 0)

          sendSupervisionPacket(SENDER\_A, REJ\_C\_0, &protocol, fd);

        else if (protocol.sequenceNumber == 1)

          sendSupervisionPacket(SENDER\_A, REJ\_C\_1, &protocol, fd);

        currentState = START;

      }

    }

    else if (buf[0] == ESCAPE)

    {

      read(fd, buf, 1);

      if (buf[0] == ESCAPEFLAG)

      {

        msg[currentIndex] = '~';

        currentIndex++;

      }

      else if (buf[0] == ESCAPEESCAPE)

      {

        msg[currentIndex] = '}';

        currentIndex++;

      }

      else

      {

        msg[currentIndex] = '}';

        currentIndex++;

        msg[currentIndex] = buf[0];

        currentIndex++;

      }

      return TRUE;

    }

    else

    {

      msg[currentIndex] = buf[0];

      currentIndex++;

      return TRUE;

    }

    break;

  default:

    break;

  }

  return FALSE;

}

int receiverReadDISC(int status, int fd)

{

  STOP = FALSE;

  unsigned char recvBuf[5];

  while (STOP == FALSE)

  {

    read(fd, recvBuf, 1);

    if (activatedAlarm)

    {

      activatedAlarm = FALSE;

      write(fd, protocol.frame, protocol.frameSize);

      alarm(protocol.timeout);

    }

    if (discStateMachine(status, recvBuf, &currentState, &currentIndex))

    {

      if (currentState == DONE)

      {

        alarm(0);

        protocol.currentTry = 0;

        currentIndex = 0;

        currentState = START;

        STOP = TRUE;

      }

    }

  }

  return 0;

}

int receiverReadUA(int status, int fd)

{

  STOP = FALSE;

  unsigned char recvBuf[5];

  while (STOP == FALSE)

  {

    read(fd, recvBuf, 1);

    if (activatedAlarm)

    {

      activatedAlarm = FALSE;

      write(fd, protocol.frame, protocol.frameSize);

      alarm(protocol.timeout);

    }

    if (UAStateMachine(recvBuf,status,&currentState,&currentIndex))

    {

      if (currentState == DONE)

      {

        alarm(0);

        protocol.currentTry = 0;

        currentIndex = 0;

        currentState = START;

        STOP = TRUE;

      }

    }

  }

  return 0;

}

int receiverDisconnect(int fd)

{

  receiverReadDISC(SENDER\_A, fd);

  sendSupervisionPacket(RECEIVER\_A, DISC\_C, &protocol, fd);

  alarm(protocol.timeout);

  protocol.currentTry = 0;

  receiverReadUA(RECEIVER\_A, fd);

  printf("Connection closed\n");

  return 1;

}

int setStateMachine(char \*buf)

{

  switch (currentState)

  {

  case START:

    if (buf[0] == FLAG)

    {

      currentState = FLAG\_RCV;

      currentIndex++;

      return TRUE;

    }

    break;

  case FLAG\_RCV:

    if (buf[0] == SENDER\_A)

    {

      currentState = A\_RCV;

      currentIndex++;

      return TRUE;

    }

    else if (buf[0] == FLAG)

    {

      currentIndex = 1;

      return TRUE;

    }

    else

    {

      currentState = START;

    }

    break;

  case A\_RCV:

    if (buf[0] == SET\_C)

    {

      currentState = C\_RCV;

      currentIndex++;

      return TRUE;

    }

    else if (buf[0] == FLAG)

    {

      currentIndex = 1;

      currentState = FLAG\_RCV;

      return TRUE;

    }

    else

    {

      currentState = START;

    }

    break;

  case C\_RCV:

    if (buf[0] == (SET\_C ^ SENDER\_A))

    {

      currentIndex++;

      currentState = BCC\_OK;

      return TRUE;

    }

    else if (buf[0] == FLAG)

    {

      currentIndex = 1;

      currentState = FLAG\_RCV;

      return TRUE;

    }

    else

    {

      currentState = START;

    }

  case BCC\_OK:

    if (buf[0] == FLAG)

    {

      currentIndex++;

      currentState = DONE;

      return TRUE;

    }

    else

    {

      currentIndex = 0;

      currentState = START;

    }

    break;

  default:

    break;

  }

  currentIndex = 0;

  return FALSE;

}

int readInfo(int fd, unsigned char \*appPacket)

{

  STOP = FALSE;

  unsigned char msg[MAX\_SIZE];

  unsigned char buf[2];

  while (STOP == FALSE)

  {

    buf[0] = '\0';

    read(fd, buf, 1);

    if (infoStateMachine(buf, fd,msg))

    {

      if (currentState == DONE)

      {

        currentState = START;

        STOP = TRUE;

      }

    }

    else

    {

      msg[0] = '\0';

    }

  }

  int ret = 0;

  for (int i = 0; i < currentIndex - 1; i++)

  {

    appPacket[i] = msg[i];

  }

  ret = currentIndex - 1;

  currentIndex = 0;

  return ret;

}

int readSET(int fd)

{

  STOP = FALSE;

  char buf[5];

  while (STOP == FALSE)

  {

    read(fd, buf, 1);

    if (setStateMachine(buf))

    {

      if (currentState == DONE)

      {

        currentIndex = 0;

        currentState = START;

        STOP = TRUE;

      }

    }

  }

  return 0;

}

void closeReader(int fd)

{

  sleep(1);

  tcsetattr(fd, TCSANOW, &oldtio);

  close(fd);

}

int openReader(char \*port)

{

  fillProtocol(&protocol, port, 1);

  int fd = open(protocol.port, O\_RDWR | O\_NOCTTY);

  if (fd < 0)

  {

    perror(protocol.port);

    exit(-1);

  }

  if (tcgetattr(fd, &oldtio) == -1)

  { /\* save current port settings \*/

    perror("tcgetattr");

    exit(-1);

  }

  bzero(&newtio, sizeof(newtio));

  newtio.c\_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;

  newtio.c\_iflag = IGNPAR;

  newtio.c\_oflag = 0;

  /\* set input mode (non-canonical, no echo,...) \*/

  newtio.c\_lflag = 0;

  newtio.c\_cc[VTIME] = 0; /\* inter-character timer unused \*/

  newtio.c\_cc[VMIN] = 1;  /\* blocking read until 1 char 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)

  {

    perror("tcsetattr");

    exit(-1);

  }

  printf("Connection established at: %s\n", port);

  return fd;

}

void setupReaderConnection(int fd)

{

  readSET(fd);

  struct sigaction psa;

  psa.sa\_handler = atende;

  sigemptyset(&psa.sa\_mask);

  psa.sa\_flags = 0;

  sigaction(SIGALRM, &psa, NULL);

  sendSupervisionPacket(SENDER\_A, UA\_C, &protocol, fd);

}

common.h

#pragma once

#include <sys/stat.h>

#include <fcntl.h>

#include <termios.h>

#include <stdio.h>

#include <stdlib.h>

#include <signal.h>

#include <unistd.h>

#include <string.h>

#define BAUDRATE B38400

#define MODEMDEVICE "/dev/ttyS1"

#define \_POSIX\_SOURCE 1 /\* POSIX compliant source \*/

#define FALSE 0

#define TRUE 1

#define FLAG 0x7E

#define SENDER\_A 0x03

#define RECEIVER\_A 0x01

#define SET\_C 0x03

#define UA\_C 0x07

#define START 0

#define FLAG\_RCV 1

#define A\_RCV 2

#define C\_RCV 3

#define BCC\_OK 4

#define DONE 5

#define INFO\_C\_0 0x00

#define INFO\_C\_1 0x40

#define DISC\_C 0x0B

#define ESCAPE 0x7D

#define ESCAPEFLAG 0x5E

#define ESCAPEESCAPE 0x5D

#define RR\_C\_0 0x05

#define RR\_C\_1 0x85

#define REJ\_C\_0 0x01

#define REJ\_C\_1 0x81

#define MAX\_SIZE 250

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\*/

  int currentTry;                /\*Número da tentativa atual em caso de falha\*/

  char frame[MAX\_SIZE\*2+7];          /\*Trama\*/

  int frameSize;

};

void fillProtocol(struct linkLayer \*protocol ,char\* port, int Ns);

int sendSupervisionPacket(char addressField, char controlByte,struct linkLayer \* protocol, int fd);

void updateSequenceNumber(struct linkLayer \*protocol);

int discStateMachine(int status, unsigned char \*buf, int \* currentState, int\* currentIndex);

int UAStateMachine(unsigned char \*buf, char aField, int \*currentState, int\*currentIndex);

common.c

#include "common.h"

int sendSupervisionPacket(char addressField, char controlByte,struct linkLayer\* protocol, int fd)

{

  unsigned char sendBuf[5];

  sendBuf[0] = FLAG;

  sendBuf[1] = addressField;

  sendBuf[2] = controlByte;

  sendBuf[3] = addressField ^ controlByte;

  sendBuf[4] = FLAG;

  write(fd, sendBuf, 5);

  protocol->frame[0] = '\0';

  memcpy(protocol->frame, sendBuf, 5);

  protocol->frameSize = 5;

  return 0;

}

int UAStateMachine(unsigned char \*buf, char aField, int \*currentState, int\*currentIndex)

{

  switch (\*currentState)

  {

  case START:

    if (buf[0] == FLAG)

    {

      \*currentState = FLAG\_RCV;

      \*currentIndex = \*currentIndex+1;;

      return TRUE;

    }

    break;

  case FLAG\_RCV:

    if (buf[0] == aField)

    {

      \*currentState = A\_RCV;

      \*currentIndex = \*currentIndex+1;;

      return TRUE;

    }

    else if (buf[0] == FLAG)

      return TRUE;

    else

    {

      \*currentIndex = 0;

      \*currentState = START;

    }

    break;

  case A\_RCV:

    if (buf[0] == UA\_C)

    {

      \*currentIndex = \*currentIndex+1;;

      \*currentState = C\_RCV;

      return TRUE;

    }

    else if (buf[0] == FLAG)

    {

      \*currentIndex = 0;

      \*currentState = FLAG\_RCV;

      return TRUE;

    }

    else

    {

      \*currentIndex = 0;

      \*currentState = START;

    }

    break;

  case C\_RCV:

    if (buf[0] == (UA\_C ^ aField))

    {

      \*currentIndex = \*currentIndex+1;;

      \*currentState = BCC\_OK;

      return TRUE;

    }

    else if (buf[0] == FLAG)

    {

      \*currentIndex = 0;

      \*currentState = FLAG\_RCV;

      return TRUE;

    }

    else

    {

      \*currentIndex = 0;

      \*currentState = START;

    }

    break;

  case BCC\_OK:

    if (buf[0] == FLAG)

    {

      \*currentIndex = \*currentIndex+1;;

      \*currentState = DONE;

      return TRUE;

    }

    else

    {

      \*currentIndex = 0;

      \*currentState = START;

    }

    break;

  default:

    break;

  }

  \*currentIndex = 0;

  return FALSE;

}

int discStateMachine(int status, unsigned char \*buf, int \* currentState, int\* currentIndex)

{

  switch (\*currentState)

  {

  case START:

    if (buf[0] == FLAG)

    {

      \*currentState = FLAG\_RCV;

      \*(currentIndex) = \*currentIndex+1;

      return TRUE;

    }

    break;

  case FLAG\_RCV:

    if (buf[0] == status)

    {

      \*currentState = A\_RCV;

      \*currentIndex = \*currentIndex+1;

      return TRUE;

    }

    else if (buf[0] == FLAG)

      return TRUE;

    else

    {

      \*currentIndex = 0;

      \*currentState = START;

    }

    break;

  case A\_RCV:

    if (buf[0] == DISC\_C)

    {

      \*currentIndex = \*currentIndex+1;

      \*currentState = C\_RCV;

      return TRUE;

    }

    else if (buf[0] == FLAG)

    {

      \*currentIndex = 0;

      \*currentState = FLAG\_RCV;

      return TRUE;

    }

    else

    {

      \*currentIndex = 0;

      \*currentState = START;

    }

    break;

  case C\_RCV:

    if (buf[0] == (DISC\_C ^ status))

    {

      \*currentIndex = \*currentIndex+1;

      \*currentState = BCC\_OK;

      return TRUE;

    }

    else if (buf[0] == FLAG)

    {

      \*currentIndex = 0;

      \*currentState = FLAG\_RCV;

      return TRUE;

    }

    else

    {

      \*currentIndex = 0;

      \*currentState = START;

    }

  case BCC\_OK:

    if (buf[0] == FLAG)

    {

      \*currentIndex = \*currentIndex+1;

      \*currentState = DONE;

      return TRUE;

    }

    else

    {

      \*currentIndex = 0;

      \*currentState = START;

    }

    break;

  default:

    break;

  }

  \*currentIndex = 0;

  return FALSE;

}

void fillProtocol(struct linkLayer \*protocol ,char\* port, int Ns){

  strcpy(protocol->port,port);

  protocol->port[strlen(port)] = '\0';

  protocol->baudRate = BAUDRATE;

  protocol->sequenceNumber = Ns;

  protocol->timeout = 1;

  protocol->numTransmissions = 3;

  protocol->frame[0] = '\0';

  protocol->frameSize = 0;

  protocol->currentTry = 0;

}

void updateSequenceNumber(struct linkLayer \*protocol)

{

  protocol->sequenceNumber = (protocol->sequenceNumber + 1) % 2;

}