Faculdade de Engenharia da Universidade do Porto

Protocolo de Ligação de Dados

Redes de Computadores

1º Trabalho laboratorial

Equipa T02G2:

Tomás Gonçalves up201806763@fe.up.pt Diogo Nunes 201808546@fe.up.pt

Sumário

O seguinte relatório foi efetuado no âmbito da unidade curricular de Redes de Computadores (RCOM). O primeiro trabalho laboratorial tem como objetivo a transferência de dados entre dois computadores através de uma ligação por porta de série RS-232.

No final do trabalho foi possível obter um programa funcional que cumpre todos os requisitos pedidos no guião, capaz não só de efetuar uma transmissão de dados, mas também de lidar com possíveis erros durante a mesma.

1 - Introdução

O primeiro trabalho laboratorial tem como principal objetivo a implementação de um protocolo de ligação de dados e de uma aplicação que permita transferir ficheiros através da porta série entre dois computadores. Seguindo as diretrizes do guião fornecido pelos docentes, o relatório permite-nos descrever a forma como este trabalho foi abordado pelo grupo, assim como os resultados obtidos.

Abordaremos, portanto, os seguintes tópicos:

- **Arquitetura**: Apresentação dos blocos funcionais e das interfaces.
- **Estrutura do código**: API, principais estruturas de dados e funções de destaque e sua relação com a arquitetura.
- **Casos de uso principais**: Identificação dos mesmos e as sequências de chamada de funções.
- **Protocolo de ligação lógica**: Identificação dos principais aspetos funcionais e descrição da estratégia de implementação destes aspectos com apresentação de extratos de código.
- **Protocolo de aplicação**: Identificação dos principais aspectos funcionais, descrição da estratégia de implementação destes aspectos com apresentação de extractos de código.
- **Validação**: Descrição dos testes efectuados com apresentação quantificada dos resultados.
- **Eficiência do protocolo de ligação de dados**: Caracterização estatística da eficiência do protocolo e explicação do protocolo Stop & Wait.
- **Conclusões**: Síntese da informação apresentada nas secções anteriores; reflexão sobre os objectivos de aprendizagem alcançados.

2 - Arquitetura

O nosso trabalho divide-se em 2 blocos: emissor e recetor. Ambos têm funções próprias de cada bloco mas também partilham algumas funções cujo objetivo é semelhante. Apesar disso, a implementação destas garante a independência dos blocos.

3 - Estrutura do código

O código está dividido em 7 ficheiros pares de código (.c e .h) e 1 *header* de constantes:

- O par de ficheiros **app** possui a função **main** do programa e que faz ***parse*** dos argumentos da linha de comandos. Possui também a caracterização da struct applicationLayer:

```
int type;              // EMISSOR | RECETOR
char port[255];        // /dev/ttySx
char filename[255];    // Nome do ficheiro com a mensagem a enviar
char destination[255]; // Pasta de destino da mensagem a receber
char file[255];        // Ficheiro
off_t filesize;        // Tamanho do ficheiro (em bytes)
```

- Os pares de ficheiros **app_emissor** e **app_recetor** possuem as funções necessárias para o envio de pacotes (app_emissor) e para a sua receção (app_recetor).
- O par de ficheiros **data_link**, que possui as funções principais do protocolo de ligação lógica: **llopen**, **llwrite**, **llread** e **llclose**. e a definição da struck linkLayer:

```
unsigned int type;             // EMISSOR | RECETOR
char port[15];                 // Dispositivo /dev/ttySx
int baudRate;                  // Velocidade de transmissão
unsigned char sequenceNumber;  // Número de sequência de trama:
unsigned int timeout;          // Valor do temporizador
unsigned int numTransmissions; // Número de tentativas em caso de falha
unsigned char frame[MAX_SIZE]; // Trama
```

- O par de ficheiros **utils**, que possui todas as funções auxiliares ao protocolo de ligação lógica, de forma a que as funções principais fossem mais legíveis e percetíveis.
- O par de ficheiros **state_machine**, que tem como objetivo confirmar o processamento dos bytes lidos pelo recetor.

- O par de ficheiros **eficciency**, que podiam ser incluídos noutros ficheiros, mas decidimos separar para uma melhor organização do código. Possui funções que facilitam a visualização para a análise estatística ao débito, tamanho dos pacotes e velocidade de transmissão.
- O header **defines.h**, que contém todas as constantes utilizadas no trabalho, para mais fácil leitura do código das funções.

4 - Casos de uso principais

Os principais casos de uso no trabalho são:

- **A interface dos blocos**: ao dar início à aplicação nos 2 blocos, é possível identificar:
    - qual das aplicações é o emissor ou o recetor;
    - qual a porta de série a usar (na maioria dos computadores das salas para o efeito na FEUP, é usada a porta /dev/ttyS0);
    - no caso do emissor, identificar o ficheiro a enviar (o objetivo deste trabalho é o sucesso do envio do ficheiro dado pinguim.gif);
    - no caso do recetor, identificar a pasta de destino do ficheiro a receber.
- **A transmissão de dados** resume o principal objetivo do trabalho: transferir um ficheiro através da porta série. A sequência de ações a ser feita é a seguinte:
    - configuração e estabelecimento da ligação entre os dois computadores, conectados pela porta série;
    - envio e receção assíncrona dos dados do ficheiro em questão;
    - término de ligação.

5 - Protocolo de ligação lógica

O protocolo de ligação lógica consiste nas funções principais deste trabalho: llopen, llwrite, llread e llclose.

- **llopen**: trata da inicialização da conexão entre PCs pela porta série, é configurado o alarm handler para a possível necessidade de retransmissões e, de acordo com o type (emissor ou recetor), é chamada a respetiva função para o estabelecimento da ligação. Se o emissor enviar a trama e não receber uma resposta do recetor dentro de 3 segundos (valor de TIMEOUT usado no trabalho), a trama é reenviada.
    - emissor: prossegue com a função emissor_SET, que envia trama SET e recebe trama UA do recetor;
    - recetor: prossegue com a função recetor_UA, que recebe trama SET do recetor e envia trama UA.

```
// Retorna identificador da ligação de dados.
// Retorna valor negativo em caso de erro.
int llopen(char *port, int type);
```

- **llwrite**: função chamada apenas pelo emissor, para o envio de tramas de informação ao recetor. Começamos por verificar se o tamanho da mensagem não ultrapassa o limite máximo (MAX_SIZE). De seguida preparamos o cabeçalho e o fim da trama de informação (cálculo de BCC2 e verificação da necessidade de stuffing), fazemos o byte stuffing dos dados, construímos a trama total para enviar e chamamos a função ciclo_write, que não faz mais do que tratar do envio da trama e verificação da necessidade da sua retransmissão (dependendo da resposta RR ou REJ do recetor).

```
// Retorna o número de caracteres escritos (dos dados e do
resultado do byte stuffing).
// Retorna valor negativo em caso de erro.
int llwrite(int fd, char *buffer, int length);
```

- **llread**: função chamada apenas pelo recetor para a receção de tramas de informação do emissor. Tal como em llwrite, também temos uma função ciclo_read que trata da leitura (byte a byte) da informação proveniente do emissor: aquando da sua leitura, é feito o respetivo byte destuffing e verificação do BCC2. Assim que a leitura termina, é enviada uma resposta ao emissor, informando do sucesso (RR) ou insucesso (REJ) da leitura da mensagem.

```
// Retorna o número de caracteres lidos (apenas dos dados, o
destuffing é feito na máquina de estados).
// Retorna valor negativo em caso de erro.
int llread(int fd, unsigned char *buffer);
```

- **llclose**: tem como objetivo terminar a ligação entre emissor e recetor:
    - o emissor envia mensagem DISC para o recetor. Em caso de não haver resposta, espera TIMEOUT segundos e volta a tentar (durante TRIES tentativas); em caso de sucesso na receção da mensagem de resposta DISC, envia uma última mensagem UA, dá tempo para o recetor receber essa trama e termina a sua execução;
    - o recetor recebe a mensagem DISC do emissor, reenvia a mensagem DISC e espera pela receção de uma última mensagem UA, e termina a sua execução.

```
// Retorna valor positivo em caso de sucesso.
// Retorna valor negativo em caso de erro.
int llclose(int fd);
```

6 - Protocolo de aplicação

Todo este protocolo resume-se ao código que temos nos pares de ficheiros app_emissor e app_recetor:

- app_emissor trata do envio dos pacotes de controlo inicial (com o nome e tamanho do ficheiro a enviar), final e pacotes de dados para o recetor: divide o ficheiro a enviar em segmentos e coloca cada segmento junto com o seu tamanho e o número de sequência do pacote.
- app_recetor trata da receção dos pacotes enviados pelo emissor: junta todos os pacotes recebidos, e reune-os na pasta de destino passada na linha de comandos, num ficheiro com o mesmo nome que o ficheiro enviado.

Esta separação de ficheiros foi feita para evidenciar que cada bloco tem as suas ações. Ainda assim, na função main, ambos começam por inicializar a conexão entre PCs (llopen) e terminam com o seu término (llclose).

7 - Validação

Para verificar o estado do nosso trabalho, fizemos alguns testes ao longo do seu desenvolvimento:

- envio de ficheiros de diferentes tamanhos (para além do ficheiro “modelo”: pinguim.gif);
- envio de ficheiros variando os valores da capacidade de ligação (baudrate);
- envio de ficheiros variando o tamanho das tramas de informação;
- interrupção da ligação da porta série por um determinado tempo;
- geração de ruído na ligação da porta série;
- envio de ficheiros variando a percentagem de erros simulados por nós.

Apenas o teste da interrupção da ligação não teve sucesso. Todos os restantes testes foram concluídos e tiveram o resultado esperado.

8 - Eficiência do protocolo de ligação de dados

Dados os testes de validação acima mencionados, chegamos a algumas conclusões já esperadas:

- **variação do Frame Error Ratio**: esta variação afeta negativamente a eficiência do programa: os erros no BCC1 resultam na ausência de resposta do recetor, ocorrendo um TIMEOUT (o programa demora mais TIMEOUT segundos a ser executado); os erros no BCC2 também afetam a eficiência mas em menor escala, uma vez que é enviada uma resposta e em vez de ocorrer um TIMEOUT, apenas é pedido ao emissor que reenvie a mensagem. Assim sendo, **quanto maior for a percentagem de erros no BCC1 e BCC2, menor é a eficiência do programa**.
- **variação do baudrate**: os tempos de execução não variam significativamente, mas podemos dizer que **quanto maior for a capacidade de ligação, menor é a eficiência do programa**.
- **variação do tamanho das tramas de informação**: com o aumento de tamanho das tramas, há mais informação enviada por pacote e menor número de pacotes no total. Com isto, a probabilidade de erros no Frame Error Ratio também diminui. Sendo assim, **quanto maior for o tamanho das tramas de informação, maior é a eficiência do programa**.

Em relação ao protocolo de **Stop & Wait**:

- é o emissor quem envia as tramas de informação e que ficam à espera de uma confirmação positiva do recetor: **acknowledgment (ACK)**.
- ao receber essa trama, o recetor envia **ACK** no caso de não ter erros e **NACK** no caso contrário (confirmação negativa: **negative acknowledgment**).
- o emissor, ao receber ACK, continua e envia a trama seguinte; ao receber NACK, reenvia a mesma trama (que tinha erros).
- quando uma trama ou uma resposta é perdida, é necessário um mecanismo de timeout, para que seja reenviada essa trama.
- para o emissor e recetor saberem quais tramas estão à espera de receber, essas tramas e as respectivas respostas são numeradas (0 ou 1) alternadamente: ACK(i) significa que o recetor está à espera de uma trama I(i).

No nosso trabalho foi usado um protocolo de Stop & Wait para deteção de erros: no envio das tramas de informação, o emissor fica sempre à espera de uma resposta positiva (sem erros: RR) ou negativa (com erros: REJ), para este saber se envia a trama seguinte ou se reenvia a mesma trama.

Trama enviada pelo emissor e resposta do recetor:

- Ns = 0 -> RR (Nr = 1: sem erros) ou REJ (Nr = 0: com erros).
- Ns = 1 -> RR (Nr = 0: sem erros) ou REJ (Nr = 1: com erros).

9 - Conclusões

Os objetivos deste trabalho passavam por implementar um protocolo de ligação de dados e testá-lo com uma aplicação simples de transferência de ficheiros.

Assim sendo, visto que conseguimos fornecer um serviço de comunicação de dados fiável entre 2 sistemas ligados por uma porta série, usando uma aplicação com uma estrutura bem definida do que faz o emissor e/ou o recetor, concluímos que o trabalho foi concluído com sucesso.

Anexo I - Código-fonte

- app.h

```h
#ifndef APP_H
#define APP_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>

#include "defines.h"
#include "app_emissor.h"
#include "app_recetor.h"
#include "data_link.h"

struct applicationLayer {
  int type;                 // EMISSOR | RECETOR
  char port[255];           // /dev/ttySx
  char filename[255];       // Nome do ficheiro com a mensagem a enviar
  char destination[255];    // Pasta de destino da mensagem a receber

  char file[255];           // Ficheiro
  off_t filesize;           // Tamanho do ficheiro (em bytes)
} app;

// Função que lê os argumentos da linha de comandos e os coloca na struct app.
void parseArguments(int argc, char **argv);

// Função main.
int main(int argc, char** argv);

#endif // APP_H
```

- app.c

```c
#include "app.h"

void parseArguments(int argc, char **argv) {
    if (argc != 4) {
        printf("Uso: ./app emissor portaSerie ficheiro\n");
        printf("Uso: ./app recetor portaSerie destino\n");
        exit(-1);
    }
    if (strcmp("emissor", argv[1]) == 0) {
        app.type = EMISSOR;
        strcpy(app.filename, argv[3]);
    }
    else if (strcmp("recetor", argv[1]) == 0) {
        app.type = RECETOR;
        strcpy(app.destination, argv[3]);
    }

    strcpy(app.port, argv[2]);

    if (app.type == EMISSOR) printf("Emissor\n\n");
    else if (app.type == RECETOR) printf("Recetor\n\n");
}

int main(int argc, char** argv) {
    printf("\n\nAplicação - RCOM - TL1\n");

    parseArguments(argc, argv);

    printf("A inicializar a conexão...\n");
    int fd = llopen(app.port, app.type);
    if (fd < 0) {
        printf("Erro em llopen()...\n");
        return -1;
    }

    if (app.type == EMISSOR) {
        if (appEmissor(fd) < 0) printf("Erro no appEmissor()...\n");
    }
    else if (app.type == RECETOR) {
        if (appRecetor(fd) < 0) printf("Erro no appRecetor()...\n");
    }

    printf("\nA terminar a conexão...\n");
    if (llclose(fd) < 0) {
        printf("Erro em llclose()\n");
        return -1;
    }

    if (app.type == EMISSOR) printf("Emissor terminou execução.\n\n");
    else if (app.type == RECETOR) printf("Recetor terminou execução.\n\n");

    return 0;
}
```

- app_emissor.h

```c
#ifndef APP_EMISSOR_H
#define APP_EMISSOR_H

#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <string.h>

#include "app.h"
#include "defines.h"
#include "eficiency.h"

struct stat ficheiro; //stat ficheiro do pinguim

// Função que trata do envio do pacote de controlo inicial.
int startControlPacket(int fd);

//Função que trata do envio do pacote de controlo final.
int endControlPacket(int fd);

// Função que trata do envio dos pacotes de controlo e de dados para o recetor.
int appEmissor(int fd);

char[] startcontrolPacket();

char[] endcontrolPacket(char[] cPack); //cPack = pack inicial



#endif // APP_EMISSOR_H
```

- app_emissor.c

```c
#include "app_emissor.h"

int startControlPacket(int fd) {
    char cPack[255]; //Pacotes de controlo para sinalizar o início e o fim da transferência do ficheiro
    off_t fsize = sizeof(ficheiro.st_size);
    cPack[0] = C_START; //campo de controlo para indicar o start
    cPack[1] = T_SIZE; //parametro, neste caso tamanho do ficheiro
    cPack[2] = fsize; //tamanho em octetos (campo V)
    memcpy(&cPack[3],&ficheiro.st_size,fsize); //valor

    cPack[fsize+3] = T_NAME; //parametro para indicar nome
    cPack[fsize+4] = strlen(app.filename); //tamanho do ficheiro em octetos
    memcpy(&cPack[fsize+5],app.filename,strlen(app.filename)); //nome do ficheiro (pinguim)

    int nbytes = llwrite(fd, cPack, fsize + 5 + strlen(app.filename));
    return nbytes;
}

int endControlPacket(int fd) {

    char cPack[255]; //Pacotes de controlo para sinalizar o início e o fim da transferência do ficheiro
    off_t fsize = sizeof(ficheiro.st_size);
    cPack[0] = C_END; //pacote de controlo final é igual ao inicial à excessão do parametro cPack[0]
    cPack[1] = T_SIZE; //parametro, neste caso tamanho do ficheiro
    cPack[2] = fsize; //tamanho em octetos (campo V)
    memcpy(&cPack[3],&ficheiro.st_size,fsize); //valor

    cPack[fsize+3] = T_NAME; //parametro para indicar nome
    cPack[fsize+4] = strlen(app.filename); //tamanho do ficheiro em octetos
    memcpy(&cPack[fsize+5],app.filename,strlen(app.filename)); //nome do ficheiro (pinguim)


    int nbytes = llwrite(fd, cPack, fsize + 5 + strlen(app.filename));
    return nbytes;
}

int appEmissor(int fd) {
    startClock();

    if (stat(app.filename, &ficheiro)<0){
        printf("Erro no stat em appEmissor()...\n");
        return -1;
    }

    int fdfile = open(app.filename,O_RDONLY); // abre o ficheiro do pinguim
    if (fdfile <0){
        printf("Erro ao abrir ficheiro em appEmissor()...\n");
        return -1;
    }

    int numStartControlPackBytes = startControlPacket(fd);
    if (numStartControlPackBytes < 0) {
        printf("Erro ao escrever pacote de controlo inicial em appEmissor()...\n");
        return -1;
    }
    else printf("Pacote de controlo final: %d bytes escritos\n", numStartControlPackBytes);


    int num = 0;
    char dPack[MAX_SIZE]; //pacote de dados
    char * file_data[MAX_SIZE];//dados do ficheiro

    int bytes_number; //número de bytes
    int numDataPack = 1;
```

```
    while((bytes_number = read(fdfile,file_data,MAX_SIZE-4)) != 0){
        // guarda conteudo em file_data
        // guarda numero de bytes em bytes_number

        dPack[0] = C_DATA; //valor 1, parametro de dados
        dPack[1] = num % 255; //numero de sequencia (modulo de 255)
        //(K = 256 * L2 + L1)
        dPack[2] = bytes_number / 256; //L2
        dPack[3] = bytes_number % 256; //L1 resto
        memcpy(&dPack[4],file_data,bytes_number); //dados

        int numDataPackBytes = llwrite(fd, dPack, bytes_number + 4);
        if (numDataPackBytes < 0){
            //escreve os dados para fd
            printf("Erro ao escrever pacote de dados em appEmissor()...\n");
            return -1;
        }
        else printf("Pacote de dados nº %d: %d bytes escritos\n", numDataPack, numDataPackBytes);

        num++;
        numDataPack++;
    }

    int numEndControlPackBytes = endControlPacket(fd);
    if (numEndControlPackBytes < 0) {
        printf("Erro ao escrever pacote de controlo final em appEmissor()...\n");
        return -1;
    }
    else printf("Pacote de controlo final: %d bytes escritos\n", numEndControlPackBytes);

    currentClock_BperSecond(ficheiro.st_size);

    return 0;
}


char[] startcontrolPacket(){

char cPack[255];

off_t fsize = sizeof(ficheiro.st_size);
cPack[0] = C_START; //campo de controlo para indicar o start
cPack[1] = T_SIZE; //parametro, neste caso tamanho do ficheiro
cPack[2] = fsize; //tamanho em octetos (campo V)
memcpy(&cPack[3],&ficheiro.st_size,fsize); //valor

cPack[fsize+3] = T_NAME; //parametro para indicar nome
cPack[fsize+4] = strlen(app.filename); //tamanho do ficheiro em octetos
memcpy(&cPack[fsize+5],app.filename,strlen(app.filename)); //nome do ficheiro (pinguim)

return cPack;


}

char[] endcontrolPacket(char[] startPack){

  cPack[0] = C_END; //pacote de controlo final é igual ao inicial à excessão do parametro cPack[0]

  return cPack;

}
```

- app_recetor.h

```h
#ifndef APP_RECETOR_H
#define APP_RECETOR_H

#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>

#include "app.h"
#include "defines.h"
#include "eficiency.h"

// Função que guarda o conteúdo do pacote de dados recebido no ficheiro (app.file).
void file_content(unsigned char *pack, int psize);

// Função que trata da receção dos pacotes de controlo e de dados do emissor.
int appRecetor(int fd);

#endif // APP_RECETOR_H
```

- app_recetor.c

```c
#include "app_recetor.h"

void file_content(unsigned char *pack, int psize){
  off_t size = 0;
  int counter = 1;
  int info;

  while (counter != psize)
  {
    if (pack[counter]==T_SIZE){ //se for parametro de tamanho
      int info = pack[counter+1]; //size
      for (int i = counter+2, k=0; i < info+counter+2; i++, k++){ //dados a partir de pack[3];
        size += pack[i] << 8*k; //vai adicionando octetos mais significativos (com shift para a esquerda)
      }
      app.filesize = size;
      counter=info+3; // salta os bytes que nao são dados (start, parametro e size)
    }

    if (pack[counter]==T_NAME){ //se for parametro de nome
      info = pack[counter+1]; //size
      for (int i =counter+2, k=0; i < info+counter+2; i++, k++){//dados a partir de pack[3]
        app.file[k] = pack[i]; //adiciona dados ao ficheiro
      }
      app.file[info+counter+2] = '\0'; //indicador de fim de ficheiro
      counter += 2 + info; //salta os bytes que nao sao de dados
    }
  }
}
```

```c
int appRecetor(int fd){
    unsigned char pack[MAX_SIZE];
    int fdfile, lido, psize;
    int counter=0, counter2;
    int move = 0; //offsets de 255
    int numDataPack = 1;

    while(1){
        lido = llread(fd,pack); //lido = tamanho lido em pack

        if(lido<0)
            printf("Erro na leitura em appRecetor()...\n");

        if (pack[0] == C_START){ //verifica se é pacote de controlo inicial
            printf("Pacote de controlo inicial: %d bytes lidos\n", lido);
            file_content(pack, lido); //guarda o conteudo em pack

            strcat(app.destination, app.file);
            fdfile = open(app.destination, O_RDWR | O_CREAT, 0777);
            continue;
        }
        else if (pack[0] == C_DATA && lido > 0) {//verifica se é pacote de dados
            printf("Pacote de dados nº %d: %d bytes lidos\n", numDataPack, lido);

            psize = pack[3] + pack[2] * 256; //(K = 256 * L2 + L1)

            if (pack[1] != counter){
                off_t offset = (pack[1] + move) * (MAX_SIZE-4);
                lseek(fdfile, offset, SEEK_SET);
            }

            write(fdfile,&pack[4], psize); //escreve conteudo no ficheiro
            numDataPack++;

            if (pack[1] != counter){
                lseek(fdfile, 0 , 4); // SEEK_HOLE
            }

            if (pack[1]== counter){
                counter++;
            }

            counter2 = counter;
            counter %= 255;

            if (counter % 255 == 0 && counter2!= 0){
                move+=255;
            }
        }

        else if (pack[0] == C_END){
            printf("Pacote de controlo final: %d bytes lidos\n", lido);
            break;
        }
    }

    if (close(fdfile)<0){
        printf("Erro ao fechar ficheiro em appRecetor()...\n");
    }

    return fd;
}
```

- data_link.h

```h
#ifndef DATA_LINK_H
#define DATA_LINK_H

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

#include "state_machine.h"
#include "defines.h"
#include "utils.h"

struct linkLayer {
  unsigned int type;             // EMISSOR | RECETOR
  char port[15];                 // Dispositivo /dev/ttySx
  int baudRate;                  // Velocidade de transmissão
  unsigned char sequenceNumber;  // Número de sequência de trama:
  unsigned int timeout;          // Valor do temporizador
  unsigned int numTransmissions; // Número de tentativas em caso de falha
  unsigned char frame[MAX_SIZE]; // Trama
};

struct linkLayer ll;             // Protocolo
struct termios oldtio;           // Para usar em llclose
volatile int fail;               // Para usar no alarme

// Retorna identificador da ligação de dados.
// Retorna valor negativo em caso de erro.
int llopen(char *port, int type);

// Retorna o número de caracteres escritos (dos dados e do resultado do byte stuffing).
// Retorna valor negativo em caso de erro.
int llwrite(int fd, char *buffer, int length);

// Retorna o número de caracteres lidos (apenas dos dados, o destuffing é feito na máquina de estados).
// Retorna valor negativo em caso de erro.
int llread(int fd, unsigned char *buffer);

// Retorna valor positivo em caso de sucesso.
// Retorna valor negativo em caso de erro.
int llclose(int fd);

#endif // DATA_LINK_H
```

- data_link.c

```c
#include "data_link.h"

int llopen(char *port, int flag) {
  int fd;

  // Inicialização da conexão
  if (llinit(&fd, port) < 0) {
    printf("Erro em llinit()...\n");
    return -1;
  }

  // Configuração do alarme
  setting_alarm_handler();

  switch (flag) {
      case EMISSOR: {
        ll.type = EMISSOR;
        return emissor_SET(fd);
        break;
      }
      case RECETOR: {
        ll.type = RECETOR;
        return recetor_UA(fd);
        break;
      }
  }
  return 0;
}
```

```c
int llwrite(int fd, char *buffer, int length) {
  if (length > MAX_SIZE) {
    printf("O tamanho da mensagem (%d) é maior do que MAX_SIZE (%d)...\n", length, MAX_SIZE);
    return -1;
  }

  // Início de trama I
  unsigned char initBuf[4];
  initBuf[0] = FLAG;
  initBuf[1] = A_EmiRec;
  initBuf[2] = C_I(ll.sequenceNumber);
  initBuf[3] = XOR(A_EmiRec, C_I(ll.sequenceNumber));

  // Fim de trama I
  unsigned char BCC2 = buffer[0];
  for (int i = 1; i < length; i++) {
    BCC2 = XOR(BCC2, buffer[i]);
  }
  unsigned char *endBuf = (unsigned char *)malloc(2);
  int endBufSize = 2;
  // Stuffing no BCC2
  if (BCC2 == 0x7E || BCC2 == 0x7D) {
    endBufSize = 3;
    endBuf = (unsigned char *)realloc(endBuf, endBufSize);
    endBuf[0] = 0x7D;
    endBuf[1] = XOR(BCC2, 0x20);
    endBuf[2] = FLAG;
  }
  else {
    endBuf[0] = BCC2;
    endBuf[1] = FLAG;
  }

  // Stuffing nos dados
  int size = length;
  unsigned char *dataBuf = (unsigned char *) malloc(size);
  for (int i = 0, j = 0; i < length; i++, j++) {
    if (buffer[i] == 0x7E || buffer[i] == 0x7D) {
      // Aumentar tamanho do buffer de dados
      size++;
      dataBuf = (unsigned char *) realloc(dataBuf, size);

      dataBuf[j+1] = buffer[i] ^ 0x20;
      dataBuf[j] = 0x7D;
      j++;
    }
    else dataBuf[j] = buffer[i];
  }

  // Construção de Trama I
  int allSize = 4 + size + endBufSize, datai = 0, endi = 0;
  unsigned char allBuf[allSize];
  for (int i = 0; i < allSize; i++) {
    if (i < 4) {
      allBuf[i] = initBuf[i];
    }
    else if (datai < size) {
      allBuf[i] = dataBuf[datai];
      datai++;
    }
    else if (endi < endBufSize) {
      allBuf[i] = endBuf[endi];
      endi++;
    }
  }

  settingUpSM(WRITE, START, A_EmiRec, C_RR(XOR(ll.sequenceNumber, 0x01)));

  if (ciclo_write(fd, allBuf, sizeof(allBuf)) < 0) {
    printf("llwrite(): Falha no ciclo write\n");
    return -1;
  }

  ll.sequenceNumber = XOR(ll.sequenceNumber, 0x01);

  free(dataBuf);
  free(endBuf);

  return sizeof(allBuf);
}
```

```c
int llread(int fd, unsigned char *buffer) {
  settingUpSM(READ, START, A_EmiRec, C_I(ll.sequenceNumber));
  unsigned char *dataBuf;
  int size;
  unsigned char cValue;

  if (ciclo_read(fd, &cValue, &dataBuf, &size) < 0) {
    printf("Erro no ciclo read\n");
    return -1;
  }

  unsigned char reply[5];
  reply[0] = FLAG;
  reply[1] = A_EmiRec;
  reply[2] = cValue;
  reply[3] = XOR(A_EmiRec, cValue);
  reply[4] = FLAG;

  int res = write(fd, reply, 5);
  if (res == -1) {
    printf("Erro a escrever resposta em llread()...\n");
    return -1;
  }

  for (int i = 0; i < size; i++) {
    buffer[i] = dataBuf[i];
  }

  return size;
}
```

```c
int llclose(int fd) {
  int ret = fd;

  switch (ll.type) {
    case EMISSOR: if (emissor_DISC(fd) == 0) ret = -1; break;
    case RECETOR: if (recetor_DISC(fd) == 0) ret = -1; break;
  }

  if ( tcsetattr(fd,TCSANOW,&oldtio) == -1) {
    perror("tcsetattr");
    return 1;
  }
  if (close(fd) != 0) {
    printf("Erro em close()\n");
    return 1;
  }

  return ret;
}
```

- utils.h

```c
#ifndef UTILS_H
#define UTILS_H

#include "data_link.h"

// Função para imprimir um valor em hexadecimal.
void print_0x(unsigned char a);

// Função do alarm handler.
void atende();

// Função que configura o alarm handler.
int setting_alarm_handler();

// Inicialização da conexão.
int llinit(int *fd, char *port);

// Função auxiliar do ciclo write em llwrite.
int ciclo_write(int fd, unsigned char *buf, int bufsize);

// Função auxiliar do ciclo read em llread.
int ciclo_read(int fd, unsigned char *c, unsigned char **dataBuf, int *size);

// Função que envia Trama SET e recebe Trama UA
int emissor_SET(int fd);

// Função que recebe Trama SET e envia Trama UA
int recetor_UA(int fd);

// Função que envia Trama DISC, recebe Trama DISC e envia Trama UA.
int emissor_DISC(int fd);

// Função que recebe Trama DISC, envia Trama DISC e recebe Trama UA.
int recetor_DISC(int fd);

#endif // UTILS_H
```

- utils.c

```c
#include "utils.h"

static struct sigaction old; // Para usar no restauro do SIGALRM

void print_0x(unsigned char a) {
  if (a == 0) printf("0x00 ");
  else        printf("%#.2x " , a);
}

void atende() {
  if (SM.state != SM_STOP) {
    fail = TRUE;
    printf("Alarm!\n");
    return;
  }
}

int setting_alarm_handler() {
  struct sigaction sa;
  sigemptyset(&sa.sa_mask);
  sa.sa_handler = atende;
  sa.sa_flags = 0;
  if (sigaction(SIGALRM, &sa, &old) < 0) {
    printf("Erro no sigaction...\n");
    return -1;
  }
  return 0;
}
```

```c
int llinit(int *fd, char *port) {

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

  strcpy(ll.port, port);
  ll.baudRate = BAUDRATE;
  ll.sequenceNumber = 0x00;
  ll.timeout = TIMEOUT;
  ll.numTransmissions = TRIES;

  struct termios newtio;

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

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

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

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

  tcflush(*fd, TCIOFLUSH);

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

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

  return 0;
}
```

```c
int ciclo_write(int fd, unsigned char *buf, int bufsize) {
  int try = 0;
  volatile int STOP = FALSE;

  do {
    try++;
    int res = write(fd, buf, bufsize);
    if (res == -1) {
      printf("Erro a enviar Trama I para o buffer no ciclo_write()...\n");
      return -1;
    }

    alarm(ll.timeout);
    SM.state = START;
    fail = FALSE;
    unsigned char readBuf[1];

    while (STOP == FALSE) {
      res = read(fd, readBuf, 1);

      if (res == -1 && errno == EINTR) {
        printf("Erro a receber RR do recetor no ciclo_write()...\n");
        if (try < ll.numTransmissions) printf("Nova tentativa...\n");
        else {
          printf("Excedeu o número de tentativas...\n");
          return -1;
        }
        break;
      }
      else if (res == -1) {
          printf("Erro a receber RR do buffer no ciclo_write()...\n");
          return -1;
      }

      if (stateMachine(readBuf[0], NULL, NULL) < 0) {
        printf("Erro a receber RR ou REJ no ciclo_write()...\n");
        fail = TRUE;
        alarm(0);
        break;
      }

      if (SM.state == SM_STOP || fail) STOP = TRUE;
    }
  } while (try < ll.numTransmissions && fail);

  alarm(0);
  return 0;
}
```

```c
int ciclo_read(int fd, unsigned char *C, unsigned char **dataBuf, int *size) {
  volatile int STOP = FALSE;
  unsigned char buf[1];

  while (STOP == FALSE) {
    int res = read(fd, buf, 1);
    if (res == -1) {
      printf("Erro a receber trama I do buffer no ciclo_read()...\n");
      return -1;
    }

    int smRead = stateMachine(buf[0], dataBuf, size);
    if (smRead == -1) {
      *C = C_REJ(ll.sequenceNumber);
      printf("Erro no BCC... no ciclo_read()...\n");
      return -1;
    }
    else if (smRead == -2) {
      *C = C_RR(ll.sequenceNumber);
      printf("Número de sequência errado em C no ciclo_read()...\n");
      return -2;
    }
    *C = C_RR(ll.sequenceNumber);

    if (SM.state == SM_STOP) STOP = TRUE;
  }
  return 0;
}
```

```c
int emissor_SET(int fd) {
  unsigned char buf[SET_UA_DISC_SIZE];
  buf[0] = FLAG;
  buf[1] = A_EmiRec;
  buf[2] = C_SET;
  buf[3] = XOR(A_EmiRec, C_SET);
  buf[4] = FLAG;

  settingUpSM(SV, START, A_RecEmi, C_UA);

  int res, num_try = 0;
  volatile int STOP=FALSE;

  do {
    num_try++;
    res = write(fd, buf, SET_UA_DISC_SIZE);
    if (res == -1) {
      printf("Erro no envio de mensagem SET...\n");
      return 1;
    }
    else {
      printf("Mensagem SET enviada: ");
      for (int i = 0; i < SET_UA_DISC_SIZE; i++) print_0x(buf[i]);
      printf("\n");
    }

    alarm(ll.timeout);
    SM.state = START;
    fail = FALSE;
    unsigned char readBuf[1];

    printf("Mensagem UA recebida: ");
    while (STOP == FALSE) {
      res = read(fd, readBuf, 1);
      if (res == -1 && errno == EINTR) {
        printf("Erro a receber a mensagem UA do recetor...\n");
        if (num_try < ll.numTransmissions) printf("Nova tentativa...\n");
        else {
          printf("Excedeu o número de tentativas...");
          return -1;
        }
        break;
      }
      else if (res == -1) {
        printf("Erro a receber a mensagem UA do buffer...\n");
        return 1;
      }
      print_0x(readBuf[0]);

      stateMachine(readBuf[0], NULL, NULL); // **buf and *size not needed here

      if (SM.state == SM_STOP || fail) STOP = TRUE;
    }
  } while (num_try < ll.numTransmissions && fail);
  printf("\n\n");

  if (fail) {
    printf("Todas as tentativas de receber UA foram sem sucesso...\n");
    return 1;
  }

  alarm(0);

  return fd;
}
```

```c
int recetor_UA(int fd) {
  settingUpSM(SV, START, A_EmiRec, C_SET);

  unsigned char buf[1];
  int res;
  volatile int STOP=FALSE;

  printf("Mensagem SET recebida: ");
  while (STOP==FALSE) {
    res = read(fd, buf, 1);
    if (res == -1) {
      printf("Erro a receber mensagem SET\n");
      return 1;
    }
    print_0x(buf[0]);

    stateMachine(buf[0], NULL, NULL); // **buf and *size not needed here

    if (SM.state == SM_STOP) STOP = TRUE;
  }
  printf("\n");

  // Resposta do recetor
  unsigned char reply[SET_UA_DISC_SIZE];
  reply[0] = FLAG;
  reply[1] = A_RecEmi;
  reply[2] = C_UA;
  reply[3] = XOR(A_RecEmi, C_UA);
  reply[4] = FLAG;

  res = write(fd, reply, SET_UA_DISC_SIZE);
  if (res == -1) {
    printf("Erro no envio de mensagem UA\n");
    return 1;
  }
  else {
    printf("Mensagem UA enviada: ");
    for (int i = 0; i < SET_UA_DISC_SIZE; i++) print_0x(reply[i]);
    printf("\n\n");
  }

  return fd;
}
```

```c
int emissor_DISC(int fd) {
  unsigned char msgDISC[SET_UA_DISC_SIZE];
  msgDISC[0] = FLAG;
  msgDISC[1] = A_EmiRec;
  msgDISC[2] = C_DISC;
  msgDISC[3] = XOR(A_EmiRec, C_DISC);
  msgDISC[4] = FLAG;

  settingUpSM(SV, START, A_RecEmi, C_DISC);

  int res, num_try = 0;
  volatile int STOP=FALSE;

  do {
    num_try++;
    res = write(fd, msgDISC, SET_UA_DISC_SIZE);
    if (res == -1) {
      printf("Erro no envio de mensagem DISC\n");
      return -1;
    }
    else {
      printf("Mensagem DISC enviada: ");
      for (int i = 0; i < SET_UA_DISC_SIZE; i++) print_0x(msgDISC[i]);
      printf("\n");
    }

    alarm(ll.timeout);
    SM.state = START;
    fail = FALSE;
    unsigned char readBuf[1];

    printf("Mensagem DISC recebida: ");
    while (STOP == FALSE) {
      res = read(fd, readBuf, 1);
      if (res == -1 && errno == EINTR) {
        printf("Erro a receber a mensagem DISC do recetor...\n");
        if (num_try < ll.numTransmissions) printf("Nova tentativa...\n");
        else {
          printf("Excedeu o número de tentativas...\n");
          return -1;
        }
        break;
      }
      else if (res == -1) {
        printf("Erro a receber a mensagem DISC do buffer...\n");
        return -1;
      }
      print_0x(readBuf[0]);

      stateMachine(readBuf[0], NULL, NULL);

      if (SM.state == SM_STOP || fail) STOP = TRUE;
    }
  } while (num_try < ll.numTransmissions && fail);
  printf("\n");

  if (fail) {
    printf("Todas as tentativas de receber DISC foram sem sucesso...\n");
    return -1;
  }

  alarm(0);

  unsigned char msgUA[SET_UA_DISC_SIZE];
  msgUA[0] = FLAG;
  msgUA[1] = A_RecEmi;
  msgUA[2] = C_UA;
  msgUA[3] = XOR(A_RecEmi, C_UA);
  msgUA[4] = FLAG;

  res = write(fd, msgUA, SET_UA_DISC_SIZE);
  if (res == -1) {
    printf("emissor_DISC(): Erro no envio de mensagem UA\n");
    return -1;
  }
  else {
    printf("Mensagem UA enviada: ");
    for (int i = 0; i < SET_UA_DISC_SIZE; i++) print_0x(msgUA[i]);
    printf("\n\n");
  }

  sleep(3);

  return fd;
}
```

```c
int recetor_DISC(int fd) {
  settingUpSM(SV, START, A_EmiRec, C_DISC);

  unsigned char readBuf[1];
  int res;
  volatile int STOP=FALSE;

  printf("Mensagem DISC recebida: ");
  while (STOP==FALSE) {
    res = read(fd, readBuf, 1);
    if (res == -1) {
      printf("Erro a receber mensagem DISC\n");
      return -1;
    }
    print_0x(readBuf[0]);

    stateMachine(readBuf[0], NULL, NULL);

    if (SM.state == SM_STOP) STOP = TRUE;
  }

  // Resposta do recetor
  unsigned char reply[SET_UA_DISC_SIZE];
  reply[0] = FLAG;
  reply[1] = A_RecEmi;
  reply[2] = C_DISC;
  reply[3] = XOR(A_RecEmi, C_DISC);
  reply[4] = FLAG;

  res = write(fd, reply, SET_UA_DISC_SIZE);
  if (res == -1) {
    printf("Erro no envio de mensagem DISC\n");
    return -1;
  }
  else {
    printf("\nMensagem DISC enviada: ");
    for (int i = 0; i < SET_UA_DISC_SIZE; i++) print_0x(reply[i]);
    printf("\n");
  }

  settingUpSM(SV, START, A_RecEmi, C_UA);
  STOP = FALSE;
  alarm(ll.timeout);

  printf("Mensagem UA recebida: ");
  while (STOP==FALSE) {
    res = read(fd, readBuf, 1);
    if (res == -1) {
      printf("Erro a receber mensagem UA\n");
      return -1;
    }
    print_0x(readBuf[0]);

    stateMachine(readBuf[0], NULL, NULL);

    if (SM.state == SM_STOP) STOP = TRUE;
  }
  printf("\n\n");

  alarm(0);

  sleep(3);

  return fd;
}
```

- state_machine.h

```h
#ifndef STATE_MACHINE_H
#define STATE_MACHINE_H

#include "data_link.h"
#include "eficiency.h"

enum stateMachineType {
    SV,     // Supervisão
    WRITE,  // Escrita
    READ    // Leitura
};

enum stateMachineState {
    START,
    FLAG_RCV,
    A_RCV,
    C_RCV,
    BCC_OK,
    SM_STOP // DONE
};

struct stateMachine {
    enum stateMachineType type;     // SV | WRITE | READ
    enum stateMachineState state;   // START | FLAG_RCV | A_RCV | C_RCV | BCC_OK | SM_STOP
    unsigned char A;                // Address Byte
    unsigned char C;                // Control Byte
} SM;

// Função para "preparar" a máquina de estados.
void settingUpSM(enum stateMachineType type, enum stateMachineState state, unsigned char A, unsigned char C);

// Função que processa um byte e atualiza o estado na máquina de estados.
int stateMachine(unsigned char byte, unsigned char **buf, int *size);

// Função auxiliar que processa um byte caso esteja no estado START.
int SM_START(unsigned char byte);

// Função auxiliar que processa um byte caso esteja no estado FLAG_RCV.
int SM_FLAG_RCV(unsigned char byte);

// Função auxiliar que processa um byte caso esteja no estado A_RCV.
int SM_A_RCV(unsigned char byte);

// Função auxiliar que processa um byte caso esteja no estado C_RCV.
int SM_C_RCV(unsigned char byte);

// Função auxiliar que processa um byte caso esteja no estado BCC_OK.
int SM_BCC_OK(unsigned char byte, unsigned char **buf, int *size);

#endif // STATE_MACHINE_H
```

- state_machine.c

```c
#include "state_machine.h"

static unsigned char checkBuf[2]; // Para usar na verificação do BCC, guardando Address e Control Byte.
static int frameIndex, wrongC;    // Variáveis auxiliares para facilitar o uso em SM.type = READ.

void settingUpSM(enum stateMachineType type, enum stateMachineState state, unsigned char A, unsigned char C) {
    SM.type = type;
    SM.state = state;
    SM.A = A;
    SM.C = C;
}

int stateMachine(unsigned char byte, unsigned char **buf, int *size) {
    if (SM.type == READ) ll.frame[frameIndex] = byte;

    switch(SM.state) {
        case START:
            return SM_START(byte);
            break;
        case FLAG_RCV:
            return SM_FLAG_RCV(byte);
            break;
        case A_RCV:
            return SM_A_RCV(byte);
            break;
        case C_RCV:
            return SM_C_RCV(byte);
            break;
        case BCC_OK:
            return SM_BCC_OK(byte, buf, size);
            break;
        case SM_STOP:
            return 0;
            break;
        default:
            printf("Valor da máquina de estados desconhecido\n");
            return -1;
            break;
    }
}

int SM_START(unsigned char byte) {
    if (SM.type == READ) {
        frameIndex = 0;
        wrongC = FALSE;
        if (byte == FLAG) {
            SM.state = FLAG_RCV;
            frameIndex++;
        }
    }
    else {
        if (byte == FLAG) SM.state = FLAG_RCV;
    }
    return 0;
}

int SM_FLAG_RCV(unsigned char byte) {
    if (byte == SM.A) {
        SM.state = A_RCV;
        checkBuf[0] = byte;
        if (SM.type == READ) frameIndex++;
    }
    else if (byte != FLAG) SM.state = START;
    return 0;
}
```

```c
int SM_A_RCV(unsigned char byte) {
    switch (SM.type) {
        case SV:
            if (byte == SM.C) {
                SM.state = C_RCV;
                checkBuf[1] = byte;
            }
            else if (byte == FLAG) SM.state = FLAG_RCV;
            else SM.state = START;
            break;

        case WRITE:
            if (byte == C_REJ(XOR(ll.sequenceNumber, 0x01))) {
                printf("C_REJ recebido\n");
                return -1;
            }
            if (byte == SM.C) {
                SM.state = C_RCV;
                checkBuf[1] = byte;
            }
            else if (byte == FLAG) SM.state = FLAG_RCV;
            else SM.state = START;
            break;

        case READ:
            if (byte == C_I(XOR(ll.sequenceNumber, 0x01))) {
                printf("Ns errado\n");
                wrongC = TRUE;
                SM.state = C_RCV;
                checkBuf[1] = byte;
                frameIndex++;
            }
            else if (byte == SM.C) {
                SM.state = C_RCV;
                checkBuf[1] = byte;
                frameIndex++;
            }
            else if (byte == FLAG) {
                SM.state = FLAG_RCV;
                frameIndex = 1;
            }
            else SM.state = START;
            break;

        default:
            return -1;
            break;
    }
    return 0;
}
```

```c
int SM_C_RCV(unsigned char byte) {

    // Eficiência do protocolo: variar FER -> gerar erros aleatórios nas tramas de informação
    if (BCC1_PROB_ERROR != 0 && SM.type == READ) random_error_BCC1(checkBuf);

    if (byte == XOR(checkBuf[0], checkBuf[1])) {
        if (SM.type == READ && wrongC) {
            printf("Este pacote já tinha sido recebido\n");
            return -2;
        }
        SM.state = BCC_OK;
        if (SM.type == READ) frameIndex++;
    }
    else if (byte == FLAG) {
        SM.state = FLAG_RCV;
        if (SM.type == READ) frameIndex = 1;
    }
    else {
        switch(SM.type) {
            case SV:
                SM.state = START;
                break;

            case WRITE:
                printf("Erro no byte BCC\n");
                return -1;
                break;

            case READ:
                printf("BCC recebido com erros1\n");
                return -1;
                break;

            default:
                return -1;
                break;
        }
    }
    return 0;
}

int SM_BCC_OK(unsigned char byte, unsigned char **buf, int *size) {
    if (SM.type != READ) {
        if (byte == FLAG) SM.state = SM_STOP;
        else SM.state = START;
    }
    else {
        frameIndex++;
        if (byte == FLAG) {

            // Eficiência do protocolo: variar FER -> gerar erros aleatórios nas tramas de informação
            if (BCC2_PROB_ERROR != 0) random_error_BCC2(ll.frame, frameIndex);

            // Eficiência do protocolo: variar T_PROP
            sleep(DELAY_T_PROP);

            // De-Stuffing
            *buf = (unsigned char *)malloc(frameIndex-4-2);
            *size = 0;

            unsigned char destuffing;
            int lesssize = 2;
            if (ll.frame[frameIndex-3] != 0x7D) destuffing= ll.frame[frameIndex-2];
            else {
                destuffing = XOR(ll.frame[frameIndex-2], 0x20);
                lesssize = 3;
            }
            for (int i = 4; i < frameIndex - lesssize; i++) {
                if (ll.frame[i] != 0x7D) {
                    (*buf)[*size] = ll.frame[i];
                }
                else {
                    (*buf)[*size] = XOR(ll.frame[i+1], 0x20);
                    i++;
                }
                (*size)++;
            }
            *buf = (unsigned char *)realloc(*buf, (*size));

            unsigned char BCC2 = (*buf)[0];
            for (int i = 1; i < *size; i++) {
                BCC2 = XOR(BCC2, (*buf)[i]);
            }

            if (destuffing == BCC2) {
                ll.sequenceNumber = XOR(ll.sequenceNumber, 0x01);
                SM.state = SM_STOP;
            }
            else {
                printf("BCC recebido com erros2\n");
                return -1;
            }
        }
    }
    return 0;
}
```

- efficiency.h

```c
#ifndef EFICIENCY_H
#define EFICIENCY_H

#include <time.h>
#include "data_link.h"

struct timespec start;
struct timespec current;

// Função auxiliar para imprimir a velocidade de transmissão.
int baudrate_number(int b);

// Função que inicia um cronómetro.
void startClock();

/**
 * Função que imprime:
 * - a velocidade de transmissão (BAUDRATE);
 * - o tamanho dos pacotes de transmissão (MAX_SIZE);
 * - o número de bytes total do ficheiro enviado (nbytes);
 * - o tempo que passou desde startClock();
 * - o débito da transmissão (bit/s).
*/
void currentClock_BperSecond(int bytes);

// Função que gera erros aleatórios em tramas de informação no BCC1.
void random_error_BCC1(unsigned char *checkBuf);

// Função que gera erros aleatórios em tramas de informação no BCC2.
void random_error_BCC2(unsigned char *frame, int size);

#endif // EFICIENCY_H
```

- efficiency.c

```c
#include "eficiency.h"

int baudrate_number(int b) {
    switch(b) {
        case B0: return 0;
        case B50: return 50;
        case B75: return 75;
        case B110: return 110;
        case B134: return 134;
        case B150: return 150;
        case B200: return 200;
        case B300: return 300;
        case B600: return 600;
        case B1200: return 1200;
        case B1800: return 1800;
        case B2400: return 2400;
        case B4800: return 4800;
        case B9600: return 9600;
        case B19200: return 19200;
        case B38400: return 38400;
        default: return -1;
    }
}

void startClock() {
    clock_gettime(CLOCK_MONOTONIC, &start);
}

void currentClock_BperSecond(int bytes) {
    clock_gettime(CLOCK_MONOTONIC, &current);
    double time = (current.tv_sec - start.tv_sec)*1000 + (current.tv_nsec - start.tv_nsec)/10e6;
    double bits = bytes / (time / 1000);

    printf("\nVelocidade de transmissão: %d\n", baudrate_number(ll.baudRate));
    printf("Tamanho dos pacotes de informação: %d\n", MAX_SIZE);
    printf("Número de bytes: %d\n", bytes);
    printf("Tempo: %f ms\n", time);
    printf("Bits por segundo: %f\n", bits);
}

void random_error_BCC1(unsigned char *checkBuf) {
    int probability = (rand() % 100) + 1;

    if (probability <= BCC1_PROB_ERROR) {
        int i = rand() % 2;
        unsigned char random = (unsigned char)(rand() % 70);
        checkBuf[i] = random;
        printf("BCC1 gerado com erros!\n");
    }
}

void random_error_BCC2(unsigned char *frame, int size) {
    int probability = (rand() % 100) + 1;

    if (probability <= BCC2_PROB_ERROR) {
        int i = (rand() % (size - 5)) + 4;
        unsigned char random = (unsigned char)(rand() % 70);
        frame[i] = random;
        printf("BCC2 gerado com erros!\n");
    }
}
```

- defines.h

```
#ifndef DEFINES_H
#define DEFINES_H

#define BAUDRATE B38400
#define TIMEOUT 3
#define TRIES 3
#define MAX_SIZE 1024

#define BCC1_PROB_ERROR 0
#define BCC2_PROB_ERROR 0
#define DELAY_T_PROP 0

#define SET_UA_DISC_SIZE 5

#define FLAG 0x7E
#define A_EmiRec 0x03
#define A_RecEmi 0x01
#define C_SET 0b00000011 // 0x03
#define C_DISC 0b00001011 // 0x0B
#define C_UA 0b00000111 // 0x07

#define C_RR(r) ((r == 1) ? 0b10000101 : 0b00000101) // 0x05 | 0x85
#define C_REJ(r) ((r == 1) ? 0b10000001 : 0b00000001) // 0x01 | 0x81
#define C_I(r) ((r== 1) ? 0b00001000 : 0b00000000) // 0x00 | 0x40

#define XOR(a,b) (a^b)

#define C_DATA 0x01
#define C_START 0x02
#define C_END 0x03
#define T_SIZE 0x00
#define T_NAME 0x01

#define EMISSOR 0
#define RECETOR 1

#define FALSE 0
#define TRUE 1

#endif // DEFINES_H
```