Faculdade de Engenharia da Universidade do Porto
2021/22

Ligação de dados

1º trabalho laboratorial

Redes de Computadores

Licenciatura em Engenharia Informática e Computação

Turma 6 grupo 7
José Macedo (up201705226)
Marta Mariz (201907020)

Índice

Sumário

Este trabalho foi realizado no âmbito da cadeira de Redes de Computadores, era pedido aos alunos a implementação de um protocolo de comunicação assíncrona para a transmissão de um ficheiro através de uma Porta Série RS-232.

Introdução

O trabalho tem como objetivo a implementação de um protocolo de ligação de dados, de acordo com as especificações fornecidas no guião de trabalho. Abaixo está por tópicos explicada a organização e execução do projeto, desenvolvido com a estrutura indicada.

Arquitetura

Este projeto adota uma arquitetura em camadas que se baseia no princípio de independência entre camadas.
A aplicação deste princípio no nosso projeto implica a existência de duas camada, a camada do protocolo de ligação de dados (Link Layer) e a camada da aplicação (Aplication Layer).

Blocos funcionais

Os ficheiros data_layer.c e data_layer.h representam a Link Layer, os ficheiros alarm.c , alarm.h, e macros.h são blocos funcionais onde se agrupam diversas funções auxiliares à Link Layer.
Os ficheiros app_layer.c, app_layer.h, app.c, app.h, reader.c, reader.h, writer.c e o writer.h representam a Aplication Layer.
Os ficheiros handle_file.c, handle_file.h e macros.h são blocos funcionais onde se agrupam diversas funções auxiliares à Aplication Layer.

Interface

Num terminal é permitido ao utilizador correr a aplicação “app” juntamente com os argumentos necessários. O primeiro argumento é referente ao Serial Port a ser usado e, no caso do emissor, o ficheiro a transmitir.

Estrutura do código

O projeto está dividido entre duas pastas, lib onde estão as header files e src onde estão os ficheiros de código .c.

A execução começa na app.c, onde a partir do número de argumentos da chamada da função se percebe se foi chamado como receptor ou emissor.

Quanto à estrutura geral, está principalmente dividido em duas camadas: a camada da aplicação e a camada da ligação de dados. Ambos o emissor e o receptor partilham o código destas duas camadas para executar o seu propósito. Em baixo vão ser definidas as principais funções em cada camada e como são utilizadas do ponto de vista do receptor e do emissor.

Executável emissor:

Camada da ligação de dados:

- llopen()- Inicia o protocolo, estabelece a ligação à porta série, envia a mensagem SET ao emissor e valida a resposta UA.
- llwrite()- Constrói as tramas de informação, faz o byte stuffing e envia cada uma. Espera e avalia a resposta do recetor e aje de acordo com esta, se for REJ reenvia, se for RR ou RRepeated passa para a próxima.
- llclose()- Fecha a ligação, envia a mensagem DISC para o receptor, espera a mensagem DISC como resposta e envia a mensagem UA de confirmação.

Camada da aplicação:

- createControlPackage()- função chamada para criar control packages, o inicial com START e o final com END, este será enviado para o llwrite para que se construa uma trama de informação
- createDataPackage()- função chamada para criar data packages com pedaços do ficheiro recebido nos argumentos, estes serão enviados para o llwrite para que se construam tramas de informação

Executável recetor:

Camada da ligação de dados:

- llopen(): Inicia o protocolo, estabelece a ligação à porta série, espera pela mensagem SET do emissor e envia a resposta UA.

- llread(): função chamada para receber tramas de informação vindas do emissor, fazer o destuffing e a validação da trama, consoante o estado desta guarda ou descarta e envia a resposta apropriada ao emissor.
- llclose(): Fecha a ligação, espera pela mensagem DISC do emissor, envia a mensagem DISC como resposta e espera pela mensagem UA de confirmação.

Camada da aplicação:

- readControlPackage()- função chamada para ler o pacote de controlo, no início e no fim do envio do ficheiro, e guarda as informações obtidas.
- readDataPackage()- função chamada para ler cada pacote de dados.

Para além destas ainda há os ficheiros handle_file.c e handle_file.h, alarme.c e alarme.h e o macros.h. O handle_file tem a função que determina o tamanho da file que vem no input e o alarme define a função que é chamada quando o alarme é ativado.

Casos de uso principais

Existem dois casos de uso, quando o programa é corrido como receptor e quando é corrido como emissor.

No caso em que o programa é executado como emissor a sequência de chamada de funções, considerando as de maior relevância é:

- Chamada da função **writer()**, onde vai ser começada a execução.
- Chamada da função **llopen()**, que vai estabelecer a ligação com a porta série.
- O ficheiro passado como argumento é lido em modo ‘rb’ e descobre-se o seu tamanho através da função **get_file_size()**.
- Cria-se o pacote de controlo através da função **createContolPackage()**, passando-lhe o nome e o tamanho do ficheiro, este é enviado através do **llwrite()**.
- O ficheiro é dividido num número de bytes pré definidos, são feitos pacotes de dados através do **createDataPackage()** e enviados por **llwrite()**, até que todo o ficheiro tenha sido enviado.
- Cria-se o pacote de controlo final com **createContolPackage()** e este é enviado também através do **llwrite()**.
- No final é utilizado o **llclose()** para fechar a ligação.
- O **llwrite()** vai receber como argumento um pacote, vai passar este para a função **writeIFrame()** que vai criar um cabeçalho e fazer byte stuffing e por fim enviar a trama para o receptor. A seguir vai ficar à espera da resposta do receptor, avaliá-la através da função **receiveRFrame()**, e consoante a resposta vai atualizar o N(r) e retornar, ou vai reenviar a trama.

No caso em que o programa é executado como recetor a sequência de chamada de funções, considerando as de maior relevância é:

- Chamada da função **reader()**, onde vai ser começada a execução.
- Chamada da função **llopen()**, que vai estabelecer a ligação com a porta série.
- Recepção do pacote de controlo inicial, vai ser chamada a função **llread()**, seguida da função **readControlPackage()** que irá processar o pacote de controlo recebido e guardar o nome e o tamanho do ficheiro.
- Cria-se um ficheiro com o nome obtido em modo ‘wb’.

- Através do llread() vão ser recebidos pacotes de dados que serão processados através da função **readDataPackage()**, e os dados retornados são escritos no ficheiro.
- Quando o llread() retornar um pacote de controlo de novo, este vai ser processado através da função **readControlPackage().**
- O **llread()** chama o **receiveIFrame()** que vai ler as tramas enviadas pelo writer, avaliá-las e retornar o tipo de resposta adequada. O recieveIFrame é auxiliado pela função **receiveRFrame()** que faz o destuffing da trama e devolve o pacote de dados correspondente por argumento. No final o llread() envia uma trama de resposta ao emissor e retorna no argumento o pacote de dados.

Protocolo de ligação lógica

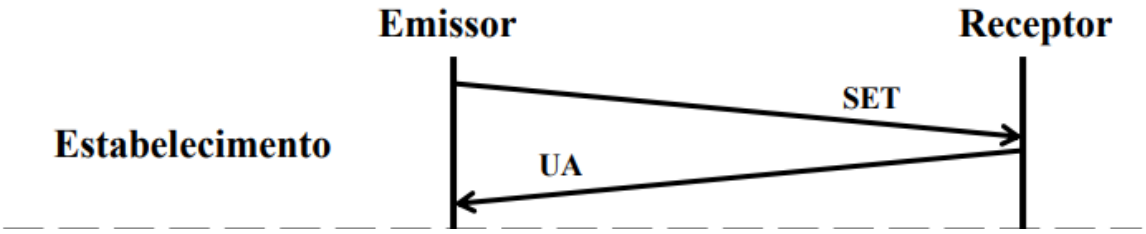
O protocolo de ligação lógica da nossa aplicação é o responsável por fornecer um serviço de comunicação de dados fiável entre dois sistemas ligados por um um cabo série.
É a camada de ligação de dados que tem a responsabilidade de interagir com a porta de série através de uma api fornecida. Nomeadamente as funções **open** e **close** que permitem a abertura e fecho da porta série, a função **write** que permite enviar tramas para a porta de série e a função **read**, que permite receber da porta de série as tramas enviadas.

Nesta camada é feita a criação das tramas de controlo, o byte stuffing antes da emissão das tramas e o destuffing quando recebe as tramas.
A emissão de tramas I, SET, e DISC são protegidas por um temporizador e em caso de ocorrência de time-out, é efectuado um número máximo de
tentativas de retransmissão.

Foram criadas nesta camada as funções previstas: **llopen**, **llclose**, **llwrite** e **llread** para servir de api para a Aplication Layer interagir com a Link Layer.

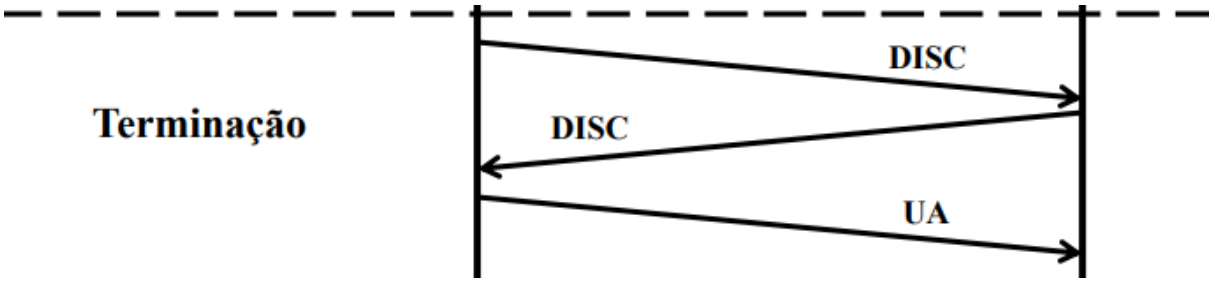
A função *int **llopen**(int porta, TRANSMITTER | RECEIVER)* recebe como argumentos a porta e a flag: TRANSMITTER / RECEIVER que identifica se a porta vai ser aberta para leitura ou escrita.
Recebendo um valor esperado na flag, a função utiliza a função open para comunicar à porta de série para abrir e configura uma nova struct termios.
Se não ocorrer nenhum erro, no caso do emissor, é enviada uma trama de controlo do tipo SET através da função write e fica a espera de receber uma resposta através da função read com uma trama de controlo do tipo UA.
No caso do receptor faz o oposto, espera pela trama de controlo SET para enviar a trama de controlo UA.
Se não conseguir estabelecer ligação retorna um valor negativo. Se a ligação for estabelecida retorna o identificador da ligação de dados.

Exemplo de uma sequência típica de tramas(sem erros):



A função *int* ***llclose****(int fd, TRANSMITTER | RECEIVER)* recebe o identificador da ligação de dados (fd) e a flag TRANSMITTER | RECEIVER como argumentos.
Recebendo os argumentos corretamente a função no caso do emissor vai enviar uma trama de controlo do tipo DISC, e esperar uma trama de controlo DISC por parte do recetor.
Quando recebe a trama de controlo DISC por parte do recetor, envia de seguida uma trama de controlo UA.
No caso do recetor, fica a aguardar a trama de controlo DISC, quando recebe esta trama envia uma trama de controlo DISC e fica a espera de receber uma trama de controlo UA.
Após a validação deste protocolo, a struct termios é reconfigurada com a struct termios anterior e termina se a ligação de dados através da função close e a função retorna um valor positivo.
No caso de não se validar o protocolo a função retorna um valor negativo.

Exemplo de uma sequência típica de tramas(sem erros):

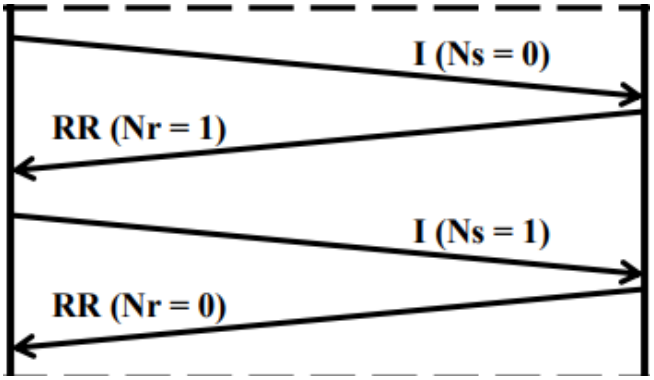


A função *int* ***llwrite****(int fd, char * buffer, int length)* recebe como argumentos o identificador da ligação de dados (fd) ,um buffer com o array de caracteres a transmitir e o comprimento do array de caracteres (length).
Recebendo os argumentos corretamente a função aplica à mensagem chegada no buffer: *framing* e *byte stuffing*, e envia essa trama de informação através da função write. De seguida, fica à espera de uma trama do tipo RR da parte do receptor.
Quando recebe uma trama do tipo RR o emissor sabe que pode mandar a próxima trama de informação e atualiza o Ns.
A função retorna o número de caracteres escritos ou um valor negativo em caso de erro.

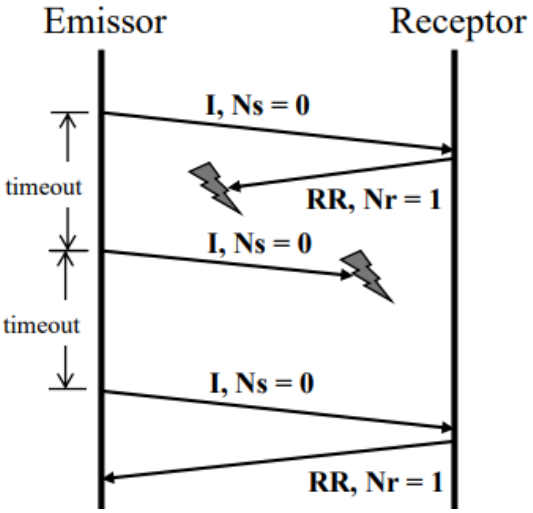
A função *int* ***llread****(int fd, char * buffer)* recebe como argumentos o identificador da ligação de dados (fd) e um buffer onde vai ficar o array de caracteres recebidos.
A função tenta ler através da função read a trama que chega a porta de série. Aplicando o deframing, destuffing e verificando se esta a ser corretamente recebida.
Se existir um erro no cabeçalho a trama é automaticamente ignorada e é enviada uma trama do tipo REJ para o emissor reenviar a trama.

Se não forem detectados erros no cabeçalho e no campo de dados as tramas são aceites para processamento.
Se se tratar de uma trama repetida, o campo de dados é descartado e é enviado uma trama do tipo RR.
Se se tratar de uma nova trama, o campo de dados é aceite ele é guardado no buffer para ser passado para a aplicação. É enviada uma trama do tipo RR, para o emissor saber que pode enviar a próxima trama e é atualizado o Nr.
A função retorna o comprimento do array (número de caracteres lidos) ou um valor negativo em caso de erro.

Exemplo de uma sequência típica de tramas(sem erros):



Exemplo de retransmissões:



Protocolo de aplicação

O protocolo de ligação é camada de mais alto nível neste código e possui quatro funções principais responsáveis por: criar o pacote de dados, ler o pacote de dados, criar o pacote de controlo e ler o este último.

- **createControlPackage**: Esta função cria o pacote de controlo, o primeiro byte sinaliza o tipo de pacote de controlo, se é START ou END. Como proposto no guião, a seguir é guardado o nome e o tamanho do ficheiro.
- **readControlPackage:** Esta função lê do pacote o nome e o tamanho do ficheiro.
- **createDataPackage:** Esta função cria o pacote de dados adicionando um cabeçalho cujo primeiro byte é 1 para indicar o seu tipo, seguido do número da sequência dos dados no ficheiro e do tamanho dos dados a enviar.
- **readDataPackage:** Esta função lê o pacote de dados, retorna o tamanho e passa os dados concretos na variável data.

Validação

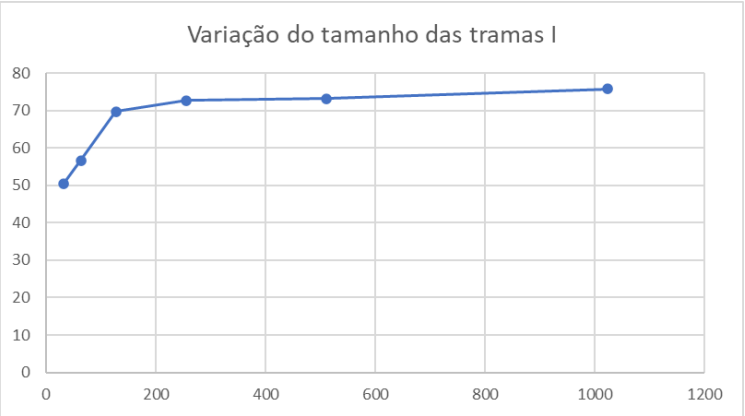
O programa desenvolvido foi submetido a testes para testar a validação do protocolo pretendido. Foram realizados constantemente testes durante o desenvolvimento. À medida que criamos funções que interagiam com a porta de série verificamos a transferência de tramas com erros para confirmar que o protocolo funcionava como pretendido. Quando passamos à criação de funções na cama da aplicação também realizamos testes para verificar a integridade do protocolo em relação à transferência de pacotes.
Foi testado a transferência de ficheiros com diferentes baudrates e diferentes tamanhos de pacotes de informação.
Durante a apresentação ao professor foram feitos testes de interrupção da comunicação na porta de série e a geração de erros de comunicação entre as portas de série.

Eficiência do protocolo de ligação de dados

No protocolo Stop & Wait o emissor transmite uma trama de informação e espera pela resposta positiva ACK (acknowledgment) pelo receptor.
O receptor recebe a trama de informação, se a trama não tiver erros confirma com a resposta ACK. No caso da trama ter erros, o receptor responde NACK.
Se o emissor recebe ACK envia a nova trama, se receber NACK retransmite a trama.
Quando as tramas de informação e as respostas ACK e NACK se perdem é necessário um timeout.

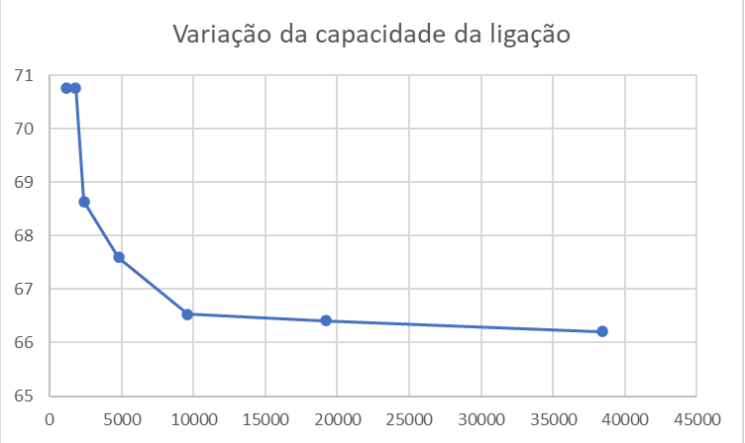
O nosso protocolo foi baseado no Stop & Wait para controlo de erros.
Quando o emissor transmite uma trama espera uma resposta do tipo RR no caso de receber os dados sem erros ou REJ no caso contrario.
Quando recebe a resposta RR o emissor sabe que deve enviar a nova trama atualizando o Ns.
O receptor recebe as tramas transmitidas pelo emissor, se forem repetidas é enviada a mensagem RR, se forem tramas novas mas tiverem erros é enviada a mensagem REJ, e Nr não é atualizado.
Quando recebe tramas novas sem erros envia RR e atualiza o Nr.

Fig 1: Estatísticas da eficiência do protocolo com a variação do tamanho das tramas.



Analisando o gráfico podemos concluir que com o aumento do tamanho do pacote, mais eficiente é a nossa aplicação. O que era de esperar pois quanto maior o pacote enviado em cada trama, menos tramas vão ser enviadas logo mais rápido o programa termina o envio do ficheiro.

Fig 2: Estatísticas da eficiência do protocolo com a variação do baudrate.



Analisando o gráfico podemos concluir que a eficiência da nossa aplicação diminui com o aumento da capacidade de ligação. O que concluímos que quanto maior o baudrate, na ocorrência de um erro, mais tramas terão de ser retransmitidas, diminuindo assim a eficiência.

Conclusões

Por fim, consideramos que fomos capazes de criar um protocolo de ligação de dados através de uma porta série de forma segura, com independência entre camadas tal como foi proposto no guião.

Compreendemos e conseguimos aplicar os conceitos teóricos da independência entre camadas, do byte stuffing e mecanismo Stop and Wait. O desenvolvimento deste trabalho laboratorial ajudou-nos a consolidar os conceitos teóricos que aprendemos e a perceber como funcionam os protocolos de rede.

Anexo - Código fonte

alarm.c

```
#include "../lib/alarme.h"

int alarmActive = FALSE, count=0;
void sig_handler(){
    printf("ALARM #%d\n", count);
    alarmActive = TRUE;
    count++;
}
```

app_layer.c

```
#include "../lib/app_layer.h"

int createControlPackage(unsigned char flag, char* fileName, int
fileSize, unsigned char* package ){

    package[0] = flag;
    package[1]= FILENAME;
    package[2] = strlen(fileName) +1 ;

    if (memcpy(&package[3] , fileName, package[2])==NULL){
        perror("Failed loading file name\n");
        return -1;
    }

    int position = 3 + strlen(fileName);

    char * length_string = (char*)malloc(sizeof(int));
    sprintf(length_string, "%d", fileSize);


    package[position] = FILESIZE;
    package[position +1 ] =  strlen(length_string) ;

    if (memcpy(&package[position + 2] , length_string,
strlen(length_string))==NULL){
```

```
        perror("Failed loading file name\n");
        return -1;
    }



    return 3 + strlen(fileName) + 2 + strlen(length_string);
}

int createDataPackage(unsigned int seqNum, unsigned int dataSize,
unsigned char * data, unsigned char * package){


    package[0] = DATA;
    package[1] = seqNum % 256;
    package[2] = dataSize  / 256;
    package[3] = dataSize  % 256;

    if (memcpy(&package[4] , data, dataSize) ==NULL){
        perror("Failed loading file name\n");
        return -1;
    }


    return 4 + dataSize;

}


int readControlPackage(unsigned char * package,  char * fileName, int*
fileSize){
    int fileNameSize;
    char fileSize_string[256];
    int index = 3; int lengthSize;
    fileName[0] = 'c';
    if (package[1] == FILENAME ){
        fileNameSize = package[2];


        if (memcpy(&fileName[1], &package[3], fileNameSize) == NULL){
            perror("Not possible to parse nameFile \n");
            return -1;
        }
```

```
    }


    index += fileNameSize -1;
    if (package[index++] == FILESIZE ){


        lengthSize = package[index++];

        if (memcpy(fileSize_string, &package[index], lengthSize) ==
NULL){
            perror("Not possible to parse nameFile \n");
            return -1;
        }


        *fileSize = atoi(fileSize_string);

    }

    return index +lengthSize;

}


int readDataPackage(unsigned char* package, int* seq,unsigned char *
data){

    *seq = package[1];
    int size = package[2]* 256 + package[3] ;


    if (memcpy(data, &package[4], size) == NULL){
        perror("Error parsing info.");
        return -1;
    }

    return size;

}
```

app.c

```
#include "../lib/app.h"


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

  if (argc > 2)
  {
      printf("Emissor\n");
      char * fileName = argv[2];
      writer(argv[1],fileName);
  }
  else
  {
      printf("Recetor\n");
      reader(argv[1]);
  }

  return 0;
}
```

data_layer.c

```
#include "../lib/data_layer.h"

extern int alarmActive, count;
struct termios oldtio,newtio;
int NS = 0, NR = 1;       // NS is send by emitter , NR is sent by
receiver
int expectedNS = 0, expectedNR = 1;  // NS expected by the receiver and
NR expected by the emitter

int writeIFrame(int fd, unsigned char *msg,  int lenght){
    unsigned char frame[FRAME_SIZE];
```

```
    int res;


    if(lenght > MAX_PACKAGE_SIZE){
        printf("Error: Data size to big to send\n");
        return -1;
    }

    frame[0] = FLAG;
    frame[1] = A_E;
    frame[2] = C_I(NS);
    frame[3] = frame[1]  ^ frame[2];

    char currentXOR = frame[3];

    int i, j;

    for(i = 4, j = 0; j < lenght; i++, j++){      //i iterates over
frame array. j iterates over msg array

        //Byte Stuffing
        if(msg[j] == FLAG){

            frame[i++] = ESC;
            frame[i] = FLAG_ESC;
        }else if(msg[j] == ESC){

            frame[i++] = ESC;
            frame[i] = ESC_ESC;
        }else {

            frame[i] = msg[j];
        }
        currentXOR = currentXOR ^ msg[j];

    }


    if(currentXOR == FLAG){

        frame[i++] = ESC;
        frame[i++] = FLAG_ESC;
    }else if(currentXOR == ESC){
```

```
            frame[i++] = ESC;
            frame[i++] = ESC_ESC;
        }else {

            frame[i++] = currentXOR;
        }

        frame[i++] = FLAG;


        res = write(fd, frame, i);




        return res;
}

int writeFrame(int fd, unsigned char A, unsigned char C){
        unsigned char frame[5];
        int res;

        frame[0] = FLAG;
        frame[1] = A;
        frame[2] = C;
        frame[3] = A ^ C;
        frame[4] = FLAG;

        res = write(fd,frame,5);


        if (res<0){
            perror("Error writing\n");
            return -1;

        }


        return 0;

}
```

```
int receiveIFrame(int fd, unsigned char *buffer)
{

    unsigned char frame;
    unsigned char bcc1;
    int status = 0;
    int res;

    while (status != 4){

        if (alarmActive) {
            break;
        }

        res = read(fd,&frame,1);
        if (res==0){
            continue;
        }

 //status machine for frame I

        switch (status){
            case 0:
                if (frame==FLAG){
                    status=1;
                }
            break;
            case 1:
                if (frame == A_E){
                    status = 2;

                }
                else if (frame !=FLAG){
                    status = 0;

                }
            break;

            case 2:
                if (frame == C_I(expectedNS)){
                    status = 3;
```

```
                }
                else if (frame == C_I(1) || frame== C_I(0)){
                    return -2; //Tramas I duplicadas
                }
                else if (frame == FLAG)
                    status = 1;
                else
                    status=0;
            break;

            case 3:
                if (frame == (C_I(expectedNS) ^ A_E)){
                    status = 4;

                    bcc1 = frame;
                    res = receiveIData(fd, bcc1, buffer);

                    return res;

                }
                else if (frame == FLAG)
                    status = 1;
                else
                    status=0;
            break;
            default:
            break;

        }
    }

    return REJ;
}

int receiveIData(int fd, unsigned char  bcc1, unsigned char  *buffer){

    int bufferSize = 0;
    int res;
    unsigned char frame = 0;

    unsigned char currentXOR = bcc1;
```

```
while(1){

    res = read(fd,&frame,1);
    if (res==0){
        continue;
    }

    if(frame == FLAG)
    {

        break;
    }
    //Byte destuffing

    if(frame == ESC){

        res = read(fd,&frame,1);
        if (res==0){
            continue;
        }

        if(frame == FLAG_ESC){
            buffer[bufferSize++] = FLAG;
        }else if(frame == ESC_ESC){
            buffer[bufferSize++] = ESC;
        }

    }else{

        buffer[bufferSize++] = frame;
    }

}


for (int i = 0; i <= bufferSize -2;i++){

    currentXOR = currentXOR ^ buffer[i];
}



if(currentXOR != buffer[bufferSize - 1]){
```

```
        printf("Frame with error, bcc2 does not match");
        return -1; //Trama I com erros
    }


    return bufferSize - 1;
}

int receiveRFrame(int fd)
{

    unsigned char frame;
      int status = 0;
      int res;
      unsigned char C = 0;
      int answerType;

      while (status != 5){

        if (alarmActive) {
          return ANSWER_TIMEOUT;
        }

        if ((res = read(fd,&frame,1))==0)
            continue;


        switch (status){
            case 0:
                if (frame==FLAG){
                    status=1;
            }
            break;
            case 1:
                if (frame == A_E){
                    status = 2;

                }
                else if (frame !=FLAG){
                    status = 0;
                }
            break;
```

```
case 2:
    if (frame == C_RR(expectedNR)){
        C = frame;
        status = 3;
        answerType = RR;
    }
    else if (frame == C_RR(0) || frame == C_RR(1) ){
        C = frame;
        status = 3;
        answerType = RR_REPEAT; // wrong C answear repeated
    }
    else if ( frame == C_REJ(expectedNR)){
        C = frame;
        status = 3;
        answerType =REJ;

    }
    else if ( frame == C_REJ(0)|| frame == C_REJ(1)){
        C = frame;
        status = 3;
        answerType =RR_REPEAT;

    }
    else if (frame == FLAG)
        status = 1;
    else
        status=0;
break;

case 3:
    if (frame == (C ^ A_E)){
        status = 4;


    }
    else if (frame== FLAG)
        status = 1;
    else
        status=0;
break;
case 4:
    if (frame == FLAG){
```

```
                    status = 5;
                }
                else
                    status=0;
            break;

            default:
            break;
            }
        }


    return answerType;


}

int recieveSFrame(int fd,unsigned char A, unsigned char C){

      unsigned char frame;
      int status = 0;
      int res;

      while (status != 5){

        if (alarmActive) {
          return ANSWER_TIMEOUT;
        }


        if ((res = read(fd,&frame,1))==0)
            continue;



        switch (status){
            case 0:
                if (frame==FLAG){
                    status=1;
            }
            break;
            case 1:
                if (frame == A){
```

```
            status = 2;

        }
        else if (frame !=FLAG){
            status = 0;
        }
    break;

    case 2:
        if (frame == C){
            status = 3;
        }
        else if (frame == FLAG)
            status = 1;
        else
            status=0;
    break;

    case 3:
        if (frame == (C ^ A)){
            status = 4;

        }
        else if (frame== FLAG)
            status = 1;
        else
            status=0;
    break;
    case 4:
        if (frame == FLAG){
          status = 5;
        }
        else
            status=0;
    break;

    default:
        break;


    }
}
```

```
    return 0;

}


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

    if (flag != TRANSMITTER && flag != RECEIVER){
        perror("Flag must be 1 or 0.");
        return -1;
    }

    if (flag == TRANSMITTER){

        fd = openPort(port);

        while (count<4 && notrecieved){
            writeFrame(fd, A_E, C_SET);
            activateAlarm();
            if ((notrecieved = recieveSFrame(fd,A_E, C_UA))==0){
                desactivateAlarm();
                break;

            }
        }
        if (notrecieved){
          perror("TIMEOUT");
          exit(-1);
        }
    }
    else if (flag == RECEIVER){
        fd = openPort(port);
        recieveSFrame(fd ,A_E ,C_SET);

        writeFrame(fd, A_E, C_UA);
    }

    return fd;

}
```

```
int openPort(char *port){

    int fd;

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

    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 fd;

}

void activateAlarm(){

    (void)signal(SIGALRM,&sig_handler);
```

```
    alarmActive = FALSE;
    alarm(3);

}

void desactivateAlarm(){

    alarm(0);
    count = 0;
    alarmActive = FALSE;
}

int llclose(int fd, unsigned char flag ){
    count = 0;
    int notrecieved = 1;

    if (flag ==TRANSMITTER){
        while (count<4 && notrecieved){
            writeFrame(fd, A_E, C_DISC);
            activateAlarm();
            if ((notrecieved = recieveSFrame(fd, A_R, C_DISC))==0){
                desactivateAlarm();
                break;
            }
        }
        if(notrecieved){
            perror("Not recieved DISC frame\n");
            exit(-1);
        }
        else{
            writeFrame(fd, A_R, C_UA);

        }

    }
    else if (flag == RECEIVER){
        while (count<4 && notrecieved){
            activateAlarm();
            if ((notrecieved = recieveSFrame(fd, A_E, C_DISC))==0){
                desactivateAlarm();
                break;
            }
        }
```

```
        if(notrecieved){
            perror("TIMEOUT: Not recieved DISC frame\n");
            exit(-1);
        }
        else{
            count = 0;
            notrecieved = 1;
            while (count<4 && notrecieved){

                writeFrame(fd, A_R, C_DISC);

                activateAlarm();
                if ((notrecieved = recieveSFrame(fd, A_R, C_UA))==0){
                    desactivateAlarm();
                    break;
                }
            }
            if(notrecieved){
                perror("TIMEOUT: Not recieved answer to DISC frame
frame\n");
                exit(-1);
            }
        }
    }


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

        close(fd);
        return 0;
}


int llwrite(int fd, unsigned char * buffer, int length)
{
    count = 0;
    int numberWrittenChars, r;

    while(count<4)
    {
```

```
            numberWrittenChars = writeIFrame(fd, buffer, length);
            activateAlarm();


            r = receiveRFrame( fd);

            if(r == RR || r == RR_REPEAT)
            {
                desactivateAlarm();
                printf("Answer RR recieved\n");
                break;
            }
            else if (r == REJ){
                desactivateAlarm();
                printf("Answer REJ recieved\n");

            }

        }
        if( count == 4){
            perror("TIMEOUT\n");
            return -1;
        }
        updateSenderN();

        return numberWrittenChars;
}

int llread(int fd, unsigned char * buffer)
{
        while(1)
        {
            int r = receiveIFrame(fd, buffer);


            if(r == -1)
            {
                writeFrame(fd, A_E, C_REJ(NR));

            }
            else if(r == -2){
                writeFrame(fd, A_E, C_RR(NR));
```

```
        }
        else if (r >= 0)
        {
            writeFrame(fd, A_E, C_RR(NR));
            updateRecieverN();

            return r;
        }

    }
}

void updateRecieverN(){
    if (expectedNS == 1) expectedNS = 0;
    else expectedNS = 1;

    if (NR == 1) NR = 0;
    else NR = 1;
}

void updateSenderN(){
    if (expectedNR == 1) expectedNR = 0;
    else expectedNR = 1;

    if (NS == 1) NS = 0;
    else NS = 1;
}
```

handle_file.c

```
#include "../lib/handle_file.h"


unsigned long get_file_size(FILE* f) {
  fseek(f, 0, SEEK_END);
  unsigned long size = ftell(f);
  fseek(f, 0, SEEK_SET);
  return size;
}
```

writer.c

```
#include "../lib/reader.h"

void reader(char *port)
{
  int fd;int res;
  int seq = 0;
  char fileName[256];
  FILE *fp;
  int length; int fileSize;
  unsigned char package[MAX_PACKAGE_SIZE];
  unsigned char data[MAX_DATA];



  fd = llopen(port, RECEIVER );



  //recieve control package

  while (TRUE){
    res = llread(fd, package);
    if (package[0]==START){
      if (readControlPackage(package, fileName, &fileSize)>0)
          break;
    }
  }



  if( (fp = fopen(fileName, "wb")) == NULL ) {
        perror("Failed creating file\n");
        exit(-1);
    }


  printf("Receiving file...\n");
  while (TRUE){
      if ( (length = llread(fd,package)) < 0){
```

```
            perror("Could not read file descriptor.");
            exit(-1);
        }


    if (package[0]== DATA){
      if ((res = readDataPackage(package, &seq, data))<0){
        exit (-1);
      };


      fwrite(data, 1, res, fp);


    }

    else if (package[0] == END){
      char endFileName[256];
      int endSize;

      if (readControlPackage(package, endFileName, &endSize)>0){

          if (strcmp(endFileName, fileName) != 0){
              perror("Incompatible information recieved\n");
              exit (-1);
          }
          if (fileSize != endSize){
              perror("Incompatible information recieved\n");
              exit(-1);
          }
          break;
      }
    }
  }



  llclose(fd, RECEIVER);

  printf("File received\n");
  return ;
}
```

reader.c

```
#include "../lib/writer.h"


void writer(char *port, char * fileName)
{
  int fd;
  unsigned char package[MAX_SIZE];
  unsigned char fileData[MAX_DATA];
  FILE *fp;
  unsigned int seq = 0;
  unsigned int informationMaxSize = MAX_DATA;
  unsigned int actualSize = 0;
  unsigned int fileDataSize = 0;

  fd = llopen(port, TRANSMITTER);//open file description and data layer
  /* open file*/
  fp = fopen(fileName, "rb");
  unsigned int fileSize = get_file_size(fp);


  int size = createControlPackage(START, fileName, fileSize, package);


  llwrite(fd, package, size);  //send control package

  printf("Sending file...\n");
  while (1){
      if (seq * informationMaxSize > fileSize){ //checks if it's not
the end of the file
        actualSize = (seq * informationMaxSize) % fileSize;
      }
      else{
        actualSize = informationMaxSize;
      }
      if (( fileDataSize = fread(fileData, 1, actualSize, fp)) == 0){
        break;
      }
```

```
      if ((size = createDataPackage(seq, fileDataSize, fileData,
package) )< 0){ // stops sending data packages if there is no more data
to be sent
        perror("Failed creating data Package\n");
        return;
      }

      if (llwrite(fd, package, size) == -1){
        return;
      }

      seq++;

  }


  size = createControlPackage(END, fileName, fileSize, package);
//send final control package
  llwrite(fd, package, size);



  llclose(fd, TRANSMITTER);

  printf("File sent\n");



  return;
}
```

alarm.h

```
#ifndef _ALARME_H
#define _ALARME_H

#include<stdio.h>
#include<unistd.h>
#include<signal.h>

#include "macros.h"


void sig_handler();

#endif
```

app_layer.h

```
#ifndef APP_LAYER_H
#define APP_LAYER_H

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "macros.h"

int createControlPackage(unsigned char flag, char* fileName, int fileSize, unsigned char*
package );

int createDataPackage(unsigned int seqNum, unsigned int dataSize, unsigned char * data,
unsigned char * package);

int readControlPackage(unsigned char * package,  char * fileName, int* fileSize);

int readDataPackage(unsigned char* package, int* seq,unsigned char * data);
#endif
```

app.h

```
#ifndef APP_H
#define APP_H

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

#include "writer.h"
#include "reader.h"

#endif
```

data_layer.h

```
#ifndef _DATA_LAYER_H
#define _DATALAYER_H


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

#include "alarme.h"

int writeIFrame(int fd, unsigned char *msg,  int lenght);
```

```
int writeFrame(int fd, unsigned char A, unsigned char C);

int receiveIFrame(int fd, unsigned char *buffer);

int receiveRFrame(int fd);

int llopen(char *port, unsigned char flag);

int openPort(char * port);

int recieveSFrame(int fd, unsigned char A, unsigned char C);

int llclose(int fd, unsigned char flag);

void activateAlarm();

void desactivateAlarm();

int receiveIData(int fd, unsigned char  bcc1, unsigned char  *buffer);

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

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

void updateSenderN();

void updateRecieverN();

#endif
```

handle_file.h

```
#ifndef HANDLE_FILE_H
#define HANDLE_FILE_H


#include <stdio.h>
#include <string.h>
#include <stdlib.h>

unsigned long get_file_size(FILE* f);
#endif
```

macros.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 A_E 0x03
#define A_R 0x01
#define C_SET 0x03
#define TIMEOUT 3
#define C_UA 0x07
#define C_DISC 0x0B
#define C_I(x) (unsigned char )(x << 6)

#define FRAME_SIZE 1024  // Maximum size of frame of data information
#define MAX_PACKAGE_SIZE (FRAME_SIZE - 6) / 2 // Maximum size of data
that can be send
#define MAX_DATA MAX_PACKAGE_SIZE-6


#define FILENAME_MAXSIZE 32


#define MAX_SIZE 1600

#define START 2
```

```
#define END 3
#define FILENAME 1
#define FILESIZE 0
#define DATA 1

#define ESC 0x7d
#define FLAG_ESC 0x5e
#define ESC_ESC 0x5d

#define TRANSMITTER 0x01
#define RECEIVER 0X00

#define C_RR(nr) (unsigned char)((nr << 7) | 0x05)
#define C_REJ(nr) (unsigned char)((nr << 7) | 0x01)

#define RR 0
#define ANSWER_TIMEOUT 1
#define REJ 2
#define RR_REPEAT 3
```

reader.h

```
#ifndef _READER_H
#define _READER_H

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

#include "data_layer.h"
#include "app_layer.h"

void reader(char *port);


#endif
```

writer.h

```
#ifndef WRITER_H
#define WRITER_H
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

#include "data_layer.h"
#include "handle_file.h"
#include "app_layer.h"

void writer(char *port, char * fileName);
#endif
```