

Redes de Computadores - L.EIC 3º ano

1º Trabalho Laboratorial

Bruno Pinheiro (201705562)

Tiago Ribeiro (202007589)

Conteúdo

Sumário	3
Introdução	3
Arquitetura	3
Estrutura do código	3
Principais casos de uso	3
Protocolo de ligação lógica	3
Protocolo de aplicação	3
Validação	3
Eficiência do protocolo de ligação de dados	4
Conclusões	4

Sumário

Este relatório foi realizado com o objetivo de complementar e detalhar o processo de elaboração do primeiro projeto laboratorial da unidade curricular de Redes de Computadores (L.EIC025) do 3º ano da Licenciatura em Engenharia Informática e Computação.

O foco deste trabalho era elaborar uma aplicação para transmitir ficheiros entre computadores através da porta de série utilizando os conhecimentos adquiridos nas aulas teóricas e laboratoriais da UC relativamente a ligações de dados.

Este foi concluído com sucesso, pelo que a aplicação foi capaz de enviar e receber o ficheiro e restabelecer a ligação após falha.

Introdução

Para este trabalho laboratorial, os alunos tinham de providenciar um serviço de comunicação de dados fiável entre dois sistemas ligados por um canal de transmissão, o cabo de série, e demonstrar a sua funcionalidade enviando um ficheiro através deste.

Assim, este relatório irá detalhar a componente teórica do projeto, estando dividido da seguinte forma:

1. Arquitetura

Apresentação dos blocos funcionais e interfaces.

2. Estrutura do código

Representação das APIs, principais estruturas de dados, principais funções e a sua relação com a arquitetura.

3. Principais casos de uso

Identificação e sequências de chamada de funções.

4. Protocolo de ligação lógica

Identificação dos principais aspectos funcionais do mesmo e descrição da estratégia de implementação destes aspectos com apresentação de extratos de código.

5. Protocolo de aplicação

Identificação dos principais aspectos funcionais do mesmo e descrição da estratégia de implementação destes aspectos com apresentação de extratos de código.

6. Validação

Descrição dos testes efetuados e apresentação quantificada 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 e reflexão sobre os objetivos de aprendizagem alcançados

Arquitetura

O trabalho está dividido em 2 blocos funcionais, o emissor e o recetor que chamam ambos funções da camada da aplicação e da camada de ligação de dados embora a estrutura interna destas permitam que o código executado diferencie consoante o papel que está a ser desempenhado.

Estrutura do código

O ficheiro principal será o main.c que chama a função applicationLayer que por sua vez chamaria as funções da camada de ligação de dados.

Principais funções da camada de ligação de dados:

- llopen: executa a ligação entre o transmissor e o recetor, o transmissor envia a trama SET e o recetor “responde” enviando a trama UA
- llwrite: envia as tramas de tipo I
- lhread: lê as tramas e envia como resposta tramas do tipo RR
- llclose: liberta o buffer, restora as definições da porta ao seu estado original e faz output de estatísticas.

Estruturas de dados:

```
typedef struct
{
    char serialPort[50];
    LinkLayerRole role;
    int baudRate;
```

```
    int nRetransmissions;
    int timeout;
} LinkLayer;
```

```
typedef struct
{
    enum state_t
    {
        S_START,
        S_FLAG,
        S_ADR,
        S_CTRL,
        S_BCC1,
        S_BCC2,
        S_DATA,
        S_ESC,
        S_END,
        S_REJ
    } state;
    unsigned char flag;
    unsigned char adr;
    unsigned char ctrl;
    unsigned char bcc;
    unsigned char *data;
    unsigned int data_size;
} Trama;
```

Variáveis globais:

LinkLayer linker;

Trama trama;

Macros pertinentes:

FLAG (0x7E)

A_SEND (0x03)

A_RESPONSE (0x01)

Principais casos de uso

Para iniciar o processo, os utilizadores devem inserir os seguintes comandos: run_tx no caso do transmissor e run_rx no recetor, estando definidos no Makefile o nome do ficheiro e as portas de série a ser utilizadas.

Assim, a transmissão de dados dá-se com a seguinte sequência:

- O utilizador, do lado do transmissor, escolhe o ficheiro a ser enviado;
- É enviada a trama de supervisão e caso não seja obtida resposta em dois ticks, a ligação é terminada;
- Caso contrário, a ligação é estabelecida;
- O transmissor envia os dados, trama a trama;
- Simultaneamente, o recetor recebe os dados, trama a trama;
- À medida que recebe os dados, o recetor guarda-os num ficheiro;
- A ligação é terminada.

Sequência de chamada de funções:

1. O main chama a função applicationLayer() com os atributos relativos à porta de série, ao seu papel, ao ficheiro
2. A função applicationLayer() chama então as funções da camada de ligação de dados (link_layer.c), começando pelo llopen()
3. Para verificar se a informação está correta, é usado o método TLV, e aí caso não sejam encontrados erros avança-se
4. Com a ligação aberta, o transmissor poderá chamar agora o llwrite para enviar informação, esperando resposta
5. Do outro lado, o recetor chama o llread e envia resposta ao transmissor.
6. O transmissor, recebendo resposta positiva, continua o processo de enviar tramas pelo llwrite até atingir o final.
7. O recetor continua a chamar o llread para receber a informação até a ter completa
8. É chamado o llclose para terminar o programa

Protocolo de ligação lógica

No protocolo de ligação lógica temos presente 4 funções: **llopen**, **llwrite**, **llread** e por fim o **llclose**.

Estas funções respetivamente fazem o estabelecimento da ligação entre o emissor e o recetor, envio de uma trama de informação, a receção de uma trama de informação e por fim finalizar a ligação entre o recetor e o emissor. É usado também uma função chamada state_machine_handler onde vai recebendo byte a byte e fazendo as mudanças de estado necessárias, de modo a que no estado final a trama esteja no formato válido.

llopen()

Esta função recebe o link layer nos seus parâmetros onde pudemos obter que role é que esse link layer tem e daí fazer a distinção. Ao fazer a distinção, o link layer com o role LITx (emissor) envia uma trama de supervisão SET e fica à espera de receber uma trama UA, ocorrendo timeout se ultrapassar o número máximo de retransmissões estabelecido. O link layer que tem o role LIRx vai ficar à espera de receber uma trama SET e só depois envia a trama UA. As trama são criadas pela função createSUFrame onde também retorna o tamanho da trama para facilitar depois percorrer byte a byte na fase a seguir. O envio e receção de trama é feito através das funções auxiliares de C que existem na biblioteca unistd.h chamadas de write and read, respetivamente. A receção das tramas é feita byte a byte e o valor de cada um é processado através da máquina de estados mencionada anteriormente.

llwrite()

O llwrite recebe um buffer e o seu tamanho e retorna 0 se tiver sucesso, caso contrário -1. Esta função é apenas usada pelo emissor onde vai criar tramas de informação e enviar para o recetor e tem várias fases:

Primeiro aloca espaço suficiente para poder escrever em memória a nova trama de informação, de seguida cria a trama de informação usando a função auxiliar criada por nós chamada createInfoFrame, que por sua vez também vai usar a técnica de byte stuffing.

Envia a trama e no final troca a variável global dataFlag de 0 para 1 ou de 1 para 0 dependendo que valor tenha.

Se no final receber um RR é executado com sucesso e sai do loop, se for RJ o número de retransmissões incrementa mais 1 e tenta fazer o processo de envio mais uma vez.

llread()

O llread recebe um packet com a data e retorna o número de caracteres lido se tiver sucesso ou -1 caso contrário. Esta função é utilizada apenas pelo recetor.

Depois de receber a trama, vai ler byte a byte e verificar o bcc2 se está correto. Se tiver correto, envia o RR com a dataFlag correta e retorna o tamanho da trama, se não envia de volta uma trama SU com o RJ. Se receber um C_DISC, este manda um RJ de volta e retorna -1 pois a operação não teve sucesso.

llclose()

O llclose envia um sinal para o alarme para então desligá-lo, liberta a memória do buffer usado e reverte a porta de série para as suas definições anteriores.

Antes de terminar, mostra no terminal estatísticas tais como o número de vezes que deu “time out”, o número de retransmissões e o tempo que levou desde o llopen ao llclose.

Protocolo de aplicação

applicationLayer()

A função applicationLayer chama a função llopen para iniciar a ligação. De seguida, usará o método TLV para enviar a informação do lado do transmissor para que, no lado do recetor, a totalidade informação (controlo, data e depois no end) seja verificada se esta está correta até ao fim. Se sim, chama as funções da camada de ligação de dados. No lado do recetor, vai verificar se está tudo sem erros, verificando package a package o primeiro elemento do array, se se tratar de C_END é porque atingiu o final, caso seja C_DATA escreverá no ficheiro. No lado do transmissor, chama então o llwrite, esperando então por resposta para que possa enviar a próxima trama, ou o llread se se tratar no lado do recetor, enviando uma resposta confirmado ter recebido a trama de seguida. Tendo sido toda a informação enviada, é chamado o llclose.

Validação

testes efetuados e resultados obtidos

Testes efetuados:

- Interrupção da ligação
- Geração de ruído na ligação
- Ficheiros de vários tamanhos e tipos

Resultados obtidos:

Estes testes foram concluídos com sucesso com exceção do último que revela a fragilidade da aplicação quando submetida a ficheiros de grande dimensão (superior a 50 KB), levando-a a falhar

Eficiência do protocolo de ligação de dados

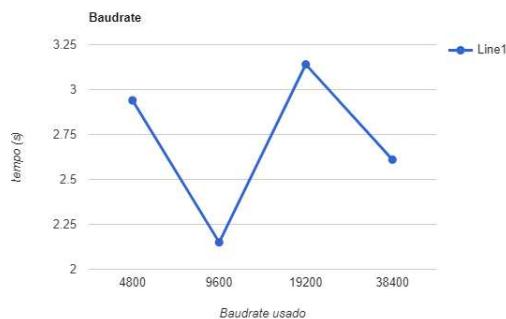
Para se avaliar a eficiência foram realizados os seguintes testes. Para cada teste foram realizados 5 ensaios para se obter valores mais fidedignos.

Estes testes terão sido realizados com o ficheiro penguin.gif (10.7 KB)

Variação do baudrate

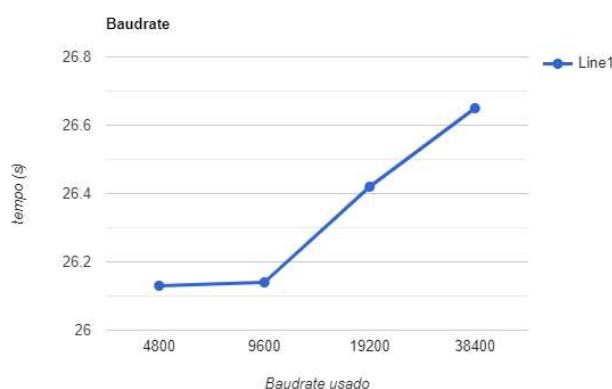
Assim, foi medido o tempo que demoravam os processos executados entre o llopen e o llclose pelo recetor e pelo transmissor.

Os resultados obtidos no lado do recetor não foram conclusivos, como vemos no gráfico seguinte



img 1. - tempo de execução do recetor de acordo com a Baudrate

No entanto, o lado do transmissor foi mais conclusivo, mostrando que existe maior eficiência nesta quando o Baudrate é menor.



img 2. - tempo de execução do transmissor de acordo com a Baudrate

Protocolo Stop & Wait:

Este protocolo consiste no seguinte mecanismo simples: depois de enviar uma trama de informação, o transmissor deve esperar pela “resposta” do recetor, o “acknowledgment” (ACK). Quando o recetor recebe a trama, verifica se esta possui erros, se não, envia ACK, se tiver, é enviada uma resposta negativa: NACK (“negative acknowledgment”). O transmissor, ao receber o ACK, continuará o processo de enviar tramas, mas se receber uma resposta negativa, terá de reenviar a trama.

Claro que pode também ocorrer erros no envio destas “respostas” logo caso o recetor demore muito tempo a enviar algo de volta, a ação dá “time out” e reenvia a trama.

Para o transmissor saber a que trama uma resposta se refere, ao mesmo tempo que também se dá ao recetor a conhecer se certa trama enviada é nova ou repetida, as tramas e os “acknowledgments” devem possuir flags que serão utilizadas alternadamente.

Na nossa aplicação, o ACK equivale ao C_RR e o NACK ao C_RJ

Conclusões

Todos os objetivos propostos para este trabalho foram atingidos.

Implementar um protocolo de ligação de dados permitiu-nos entender melhor este, nomeadamente a estrutura das tramas e a importância da independência entre camadas tal que a camada da aplicação não tem acesso a quaisquer detalhes da camada de ligação de dados, e a camada de ligação não acede a nenhum processo da camada de aplicação. Assim, a camada de aplicação trata apenas da criação e interpretação dos dados enquanto a camada de ligação apenas trata das tramas e do processo de enviar e receber informação.

Anexo:

application_layer.h:

```
#ifndef _APPLICATION_LAYER_H_
#define _APPLICATION_LAYER_H_

// Application layer main function.

// Arguments:
//   serialPort: Serial port name (e.g., /dev/ttyS0).
//   role: Application role {"tx", "rx"}.
//   baudrate: Baudrate of the serial port.
//   nTries: Maximum number of frame retries.
//   timeout: Frame timeout.
//   filename: Name of the file to send / receive.

void applicationLayer(const char *serialPort, const char *role, int baudRate,
                      int nTries, int timeout, const char *filename);

#endif // _APPLICATION_LAYER_H_
```

application_layer.c:

```
#include "application_layer.h"
#include "link_layer.h"
```

```

#include "helper.h"

#include "macros.h"

#include <string.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

unsigned char app[BUF_SIZE];

/**

 * @brief Application Layer

 *

 * @param serialPort

 * @param role

 * @param baudRate

 * @param nTries

 * @param timeout

 * @param filename

 */

void applicationLayer(const char *serialPort, const char *role, int baudRate,
int nTries, int timeout, const char *filename)

{

    LinkLayer l_layer;

    strcpy(l_layer.serialPort, serialPort);

    l_layer.baudRate = baudRate;

    l_layer.nRetransmissions = nTries;

```

```
l_layer.timeout = timeout;

if (strcmp(role, "tx") == 0)
{ // check if transmitter
    printf("tx \n");
    l_layer.role = LlTx;
    if (llopen(l_layer) == -1)
    {
        fprintf(stderr, "Couldn't open link layer\n");
        exit(1);
    }

    printf("-----Link Layer open\n");

FILE *fp = fopen(filename, "r");

if (fp == NULL)
{
    perror("Error opening the file.\n");
    exit(1);
}
else
{
    printf("File has been opened!\n");
}

fseek(fp, 0L, SEEK_END);
long int len = ftell(fp);
```

```

printf("Total size of the file = %lu bytes\n", len);

fseek(fp, 0, SEEK_SET);

app[0] = CTRL_START;

app[1] = T_SIZE;

app[2] = sizeof(long);

printf("-----Link Layer Write\n");

*((long*)(app + 3)) = len;

int flag = 0;

if (llwrite(app, 5) == -1)

{
    printf("Error llwrite\n");

    flag = 1;
}

unsigned long bytesRead = 0;

for (unsigned char i = 0; bytesRead < len && flag == 0; i++)

{
    unsigned long fileBytes = fread(app + 4, 1, (len - bytesRead <
BUF_SIZE ? len - bytesRead : BUF_SIZE), fp);

    if (fileBytes != (len - bytesRead < BUF_SIZE ? len - bytesRead :
BUF_SIZE))

    {
        flag = 1;
    }
}

```

```
        break;

    }

    app[0] = CTRL_DATA;
    app[1] = i;
    app[2] = fileBytes >> 8;
    app[3] = fileBytes % 256;

    if (llwrite(app, fileBytes + 4) == -1)
    {
        printf("Error llwrite\n"); // debugging
        flag = 1;
        break;
    }

    printf("File sent %i!\n", i);
    bytesRead += fileBytes;
}

if (!flag)
{
    app[0] = CTRL_END;
    if (llwrite(app, 1) == -1)
    {
        printf("Error llwrite\n"); // debugging
    }
    else
    {
```

```

        printf("Done!\n"); // debugging

    }

}

fclose(fp);

}

else

{

    l_layer.role = LlRx;

    printf("rx \n");

    if (llopen(l_layer) == -1)

    {

        fprintf(stderr, "Couldn't open link layer\n"); // debugging

        exit(1);

    }

    printf("Link Layer open\n");



    unsigned int len = 0, fileReceived = 0;



    printf("-----Link Layer Read\n");



    int bytesRead = llread(app);

    printf("bytes read %d \n", bytesRead); // debugging



    unsigned char type, length, *value;





    if (app[0] == CTRL_START)

    {

```

```

int jump = 1;

while (jump < bytesRead)
{

    jump += next_tlv(app + jump, &type, &length, &value);

    if (type == T_SIZE)
    {

        len = *((unsigned int *)value);

        printf("File Size: %d\n", len); // debugging
    }
}

FILE *fp = fopen(filename, "w");

if (fp == NULL)
{
    perror("Error creating the file.\n"); // debugging
    exit(1);
}

printf("File has been created!\n"); // debugging

int reachedEND = 1;

unsigned char lastNumber = 0;

while (fileReceived < len)
{
    int numberBytes = llread(app);

```

```
if (numberBytes < 1)

{
    if (numberBytes == -1)

    {
        printf("Error llread\n"); // debugging
        break;
    }

    else

    {
        printf("Error package too small\n"); // debugging
    }
}

if (app[0] == CTRL_END)

{
    printf("Disconnected!\n");
    reachedEND = 0;
    break;
}

if (app[0] == CTRL_DATA)

{
    if (app[1] != lastNumber)

    {
        printf("Error!\n"); // debugging
    }
}
```

```

        else

    {

        unsigned int size = app[3] + app[2] * 256;

        fwrite(app + 4, 1, size, fp);

        fileReceived += size;

        lastNumber++;

    }

}

}

if (reachedEND)

{

    int numberBytes = llread(app);

    if (numberBytes <= 0)

    {

        printf("Error with llread\n"); // debugging

    }

    if (app[0] == CTRL_END)

    {

        printf("Disconnecting, done!\n");

    }

    else

    {

        printf("Error received wrong package!\n"); // debugging

        for (unsigned int i = 0; i < 10; i++)

            printf("%i ", app[i]);

        printf("\n");

    }

}

```

```

        }

    }

    fclose(fp);

}

else

{

    printf("Didn't start with start package\n"); // debugging

    for (unsigned int i = 0; i < 10; i++)

        printf("%i ", app[i]);

    printf("\n");

}

printf("-----Link Layer Close\n");

llclose(1);

}

```

link_layer.h

```

#ifndef _LINK_LAYER_H_

#define _LINK_LAYER_H_


typedef enum

{
    LLTx,
    LLRx,
} LinkLayerRole;

```

```
typedef struct

{
    char serialPort[50];

    LinkLayerRole role;

    int baudRate;

    int nRetransmissions;

    int timeout;

} LinkLayer;

// SIZE of maximum acceptable payload.

// Maximum number of bytes that application layer should send to link layer

#define MAX_PAYLOAD_SIZE 1000

// MISC

#define FALSE 0

#define TRUE 1

// Open a connection using the "port" parameters defined in struct linkLayer.

// Return "1" on success or "-1" on error.

int llopen(LinkLayer connectionParameters);

// Send data in buf with size bufferSize.

// Return number of chars written, or "-1" on error.

int llwrite(const unsigned char *buf, int bufferSize);

// Receive data in packet.

// Return number of chars read, or "-1" on error.

int llread(unsigned char *packet);
```

```
// Close previously opened connection.  
  
// if showStatistics == TRUE, link layer should print statistics in the  
console on close.  
  
// Return "1" on success or "-1" on error.  
  
int llclose(int showStatistics);  
  
#endif // _LINK_LAYER_H_
```

link_layer.c:

```
#include "helper.h"  
  
// includes  
  
#include <stdio.h>  
  
#include <string.h>  
  
#include <fcntl.h>  
  
#include <unistd.h>  
  
#include <termios.h>  
  
#include <stdlib.h>  
  
#include <signal.h>  
  
#include <stdbool.h>  
  
#include <time.h>  
  
  
#define BUFFER_LIMIT (128)  
  
  
int fd;  
struct termios oldtio;  
struct termios newtio;  
LinkLayer linker;
```

```
unsigned char buffer[BUFFER_LIMIT];

unsigned char *bigBuffer;

int bigBufferSize = 0;

int dataFlag = 0;

int timeoutStats = 0;

int retransmitionsStats = 0;

clock_t start, end;

int gotDISC = 0;

// Alarm Setup

int alarmEnabled = FALSE;

int alarmCount = 0;

// Alarm function handler

void alarmHandler(int signal)

{

    alarmEnabled = FALSE;

    alarmCount++;

}

// MISC

#define _POSIX_SOURCE 1 // POSIX compliant source

///////////////////////////////

// LLOPEN

///////////////////////////////

int llopen(LinkLayer connectionParameters)
```

```

{

    printf("LLOPEN\n");

    start = clock();

    // TODO

    linker = connectionParameters;

    fd = open(linker.serialPort, O_RDWR | O_NOCTTY);

    if (fd < 0)

    {

        perror(linker.serialPort);

        exit(-1);

    }

    // Save current port settings

    if (tcgetattr(fd, &oldtio) == -1)

    {

        perror("tcgetattr");

        exit(-1);

    }

    // Clear struct for new port settings

    memset(&newtio, 0, sizeof(newtio));

    newtio.c_cflag = linker.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] = 0; // Blocking read until 5 chars received


// VTIME e VMIN should be changed in order to protect with a
// timeout the reception of the following character(s)


// Now clean the line and activate the settings for the port

// tcflush() discards data written to the object referred to
// by fd but not transmitted, or data received but not read,
// depending on the value of queue_selector:

//    TCIFLUSH - flushes data received but not read.

tcflush(fd, TCIOFLUSH);

// Set new port settings

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

(void)signal(SIGALRM, alarmHandler);


if (linker.role == LlTx)
{
    int flag = 0;

    trama.state = S_START;
}

```

```

alarmCount = 0;

while (alarmCount < linker.nRetransmissions && !flag)

{

    alarm(linker.timeout);

    alarmEnabled = TRUE;

    if (alarmCount > 0)

    {

        alarmCount++;

        printf("Alarm Count = %d\n", alarmCount);

    }

    int size = createSUFrame(buffer, A_SEND, C_SET);

    write(fd, buffer, size);

}

while (alarmEnabled && !flag)

{

    int bytes = read(fd, buffer, BUFFER_LIMIT);

    if (bytes < 0)

        return -1;

    for (int i = 0; i < bytes && !flag; i++)

    {

        state_machine_handler(&trama, buffer[i]);

        if (trama.state == S_END && trama.adr == A_SEND &&

trama.ctrl == C_UA)

            flag = 1;

    }

}

```

```

    }

    if (flag)
        printf("UA received successfully!\n"); // debugging

    else
        return -1;

    return 1;
}

else // receiver
{
    int flag = 0;

    trama.state = S_START;

    while (!flag)
    {
        int bytes = read(fd, buffer, BUFFER_LIMIT);

        if (bytes < 0)
            return -1;

        for (int i = 0; i < bytes && !flag; i++)
        {
            state_machine_handler(&trama, buffer[i]);

            if (trama.state == S_END && trama.adr == A_SEND && trama.ctrl
== C_DISC)
            {
                printf("Disconnected!\n"); // debugging

                return -1;
            }
        }
    }
}

```

```

        if (trama.state == S_END && trama.adr == A_SEND && trama.ctrl
== C_SET)

            flag = 1;

        }

    }

    if (flag)

        printf("SET received successfully!\n"); // debugging

    int tramaSize = createSUFrame(buffer, A_SEND, C_UA);

    write(fd, buffer, tramaSize);

    return 1;

}

return -1;

}

///////////////////////////////
// LLWRITE
///////////////////////////////

int llwrite(const unsigned char *buf, int bufSize)

{
    printf("LLWRITE\n"); // debugging

    if (bigBufferSize < bufSize * 2 + 10)

    {

        if (bigBufferSize == 0)

            bigBuffer = malloc(bufSize * 2 + 10);

        else

            bigBuffer = realloc(bigBuffer, bufSize * 2 + 10);

    }

}

```

```

    int tramaSize = createInfoFrame(bigBuffer, buf, bufSize, A_SEND,
C_DATA_(dataFlag));

printf("Trama size %d\n", tramaSize);

for (unsigned int i = 0; i < tramaSize;)
{
    int bytes = write(fd, bigBuffer + i, tramaSize - i);

    if (bytes == -1)

        return -1;

    i += bytes;
}

int gotPacket = 0;
int sendAgain = 0;
int retransmitions = 0;

trama.data = NULL;
alarmEnabled = TRUE;

alarm(linker.timeout);

while (!gotPacket)
{
    if (!alarmEnabled)

    {
        sendAgain = 1;

        alarmEnabled = TRUE;
        alarm(linker.timeout);
}

```

```

    }

    if (sendAgain) // checks if the package wasn't sent

    {

        if (retransmitions > 0)

            timeoutStats++;

        if (retransmitions == linker.nRetransmissions)

            return -1;

        for (unsigned int i = 0; i < tramaSize;)

        {

            int bytesWritten = write(fd, bigBuffer + i, tramaSize - i);

            if (bytesWritten == -1)

                return -1;

            i += bytesWritten;

        }

        printf("Send again\n");

        sendAgain = 0;

        retransmitions++;

    }

    int byteLido = read(fd, buffer, BUFFER_LIMIT);

    if (byteLido < 0)

        return -1;

    for (unsigned int i = 0; i < byteLido && !gotPacket && alarmEnabled;
++i)

{

```

```

state_machine_handler(&trama, buffer[i]);

printf("Current State -->%d\n", trama.state);

if (trama.state == S_END)

{

    if (trama.adr == A_SEND && (trama.ctrl == C_RR_(0) ||
trama.ctrl == C_RR_(1)))

    {

        gotPacket = 1;

        sendAgain = 0;

    }

    if (trama.adr == A_SEND && trama.ctrl == C_RJ_(dataFlag))

    {

        retransmitions = 0;

        retransmitionsStats++;

    }

}

}

if (dataFlag == 0)

{

    dataFlag = 1;

}

else

{

    dataFlag = 0;

}

return 0;
}

```

```
//////////  
// LLREAD  
/////////  
  
int llread(unsigned char *packet)  
{  
    printf("LLREAD\n"); // debugging  
  
    if (bigBufferSize < BUFFER_LIMIT)  
    {  
        if (bigBufferSize == 0)  
            bigBuffer = malloc(BUFFER_LIMIT);  
        else  
            bigBuffer = realloc(bigBuffer, BUFFER_LIMIT);  
    }  
  
    int receivedPacket = 0;  
    trama.data = packet;  
  
    while (!receivedPacket)  
    {  
  
        int bytesRead = read(fd, bigBuffer, BUFFER_LIMIT);  
  
        if (bytesRead < 0)  
            return -1;  
  
        for (unsigned int i = 0; i < bytesRead; ++i)
```

```

    {

        state_machine_handler(&trama, bigBuffer[i]);

        if (trama.state == S_REJ && trama.adr == A_SEND)

        {

            int frameSize = createSUFrame(buffer, A_SEND, (trama.ctrl ==
C_DATA_(0) ? C_RJ_(0) : C_RJ_(1)));




            write(fd, buffer, frameSize);

        }






        if (trama.state == S_END && trama.adr == A_SEND)

        {






            if (trama.ctrl == C_SET)

            {

                int frameSize = createSUFrame(buffer, A_SEND, C_UA);

                write(fd, buffer, frameSize);

            }

            if (trama.ctrl == C_DATA_(dataFlag))

            {






                if (dataFlag == 0)

                {

                    dataFlag = 1;

                }

                else

                {

                    dataFlag = 0;

                }

            }







        }

    }

}

```

```

        int frameSize = createSUFrame(buffer, A_SEND,
C_RR_(dataFlag));

        write(fd, buffer, frameSize);

        printf("Sent RR %i\n", dataFlag); // debugging to know if
the dataFlag was switching

        return trama.data_size;

    }

    else

    {

        int frameSize = createSUFrame(buffer, A_SEND,
C_RR_(dataFlag));

        write(fd, buffer, frameSize);

    }

}

if (trama.ctrl == C_DISC)

{

    gotDISC = 1;

    int frameSize = createSUFrame(buffer, A_SEND, (trama.ctrl ==
C_DATA_(0) ? C_RJ_(0) : C_RJ_(1)));

    write(fd, buffer, frameSize);

    printf("DISCONNECTED\n"); // debugging

    return -1;

    break;

}

}

return 0;
}

```

```

///////////
// LLCLOSE
///////////

int llclose(int showStatistics)
{
    printf("LLCLOSE\n");

    end = clock();

    double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
    (void)signal(SIGALRM, alarmHandler);

    if (bigBufferSize > 0)
        free(bigBuffer);

    // Restore the old port settings
    if (tcsetattr(fd, TCSANOW, &oldtio) == -1)
    {
        perror("tcsetattr");
        exit(-1);
    }

    printf("STATISTICS FOR TRANSMITTER: \n");
    printf("TIME OUT : %d\n", timeoutStats);
    printf("RETRANSMISSIONS : %d\n", retransmitionsStats);
    printf("\n");
    printf("Time taken on this side: %f\n", time_taken);

    return 1;
}

```

helper.h:

```
#ifndef _HELPER_H_
#define _HELPER_H_

#include "macros.h"
#include "link_layer.h"
#include <stddef.h>
#include <stdio.h>

typedef struct
{
    enum state_t
    {
        S_START,
        S_FLAG,
        S_ADR,
        S_CTRL,
        S_BCC1,
        S_BCC2,
        S_DATA,
        S_ESC,
        S_END,
        S_REJ
    } state;
    unsigned char flag;
    unsigned char adr;
    unsigned char ctrl;
```

```

    unsigned char bcc;
    unsigned char *data;
    unsigned int data_size;
} Trama;

Trama trama;

int createInfoFrame(unsigned char *buffer, const unsigned char *data, unsigned
int data_size, unsigned char address, unsigned char control);

int createSUFrame(unsigned char *buffer, unsigned char address, unsigned
control);

unsigned char createBCC_header(unsigned char address, unsigned char control);

unsigned char createBCC2(unsigned char *frame, int length);

int byteStuffing(const unsigned char *frame, int sizeBuffer, unsigned char
*data, unsigned char *bcc);

int next_tlv(unsigned char *buf, unsigned char *type, unsigned char *length,
unsigned char **value);

void state_machine_handler(Trama *trama, unsigned char byte);

#endif

```

helper.c:

```
#include "helper.h"
```

```

///////////
// HELPER FUNCTIONS
///////////

/** 
 * @brief Create a Information frame
 *
 * @param buffer
 * @param data
 * @param data_size
 * @param address
 * @param control
 * @return int
 */
int createInfoFrame(unsigned char *buffer, const unsigned char *data, unsigned
int data_size, unsigned char address, unsigned char control)
{
    buffer[0] = FLAG;
    buffer[1] = address;
    buffer[2] = control;
    buffer[3] = createBCC_header(address, control);

    int added_length = 0;
    unsigned char bcc = 0;

    for (int i = 0; i < data_size; i++)
        added_length += byteStuffing(data + i, 1, buffer + added_length + 4,
&bcc);
}

```

```

    added_length += byteStuffing(&bcc, 1, buffer + added_length + 4, NULL);

    buffer[added_length + 4] = FLAG;

}

/**

 * @brief creates SU frame

 *

 * @param buffer

 * @param address

 * @param control

 * @return int

 */

int createSUFrame(unsigned char *buffer, unsigned char address, unsigned
control)

{
    buffer[0] = FLAG;

    buffer[1] = address;

    buffer[2] = control;

    buffer[3] = createBCC_header(address, control);

    buffer[4] = FLAG;

    return 5;
}

/**

 * @brief creates bcc1

```

```

/*
 * @param address
 * @param control
 * @return unsigned char
 */

unsigned char createBCC_header(unsigned char address, unsigned char control)
{
    return address ^ control;
}

/***
 * @brief inserts noninformation bits into data to be transferred
 *
 * @param frame
 * @param sizeBuffer
 * @param data
 * @param bcc
 * @return int
 */
int byteStuffing(const unsigned char *frame, int sizeBuffer, unsigned char
*data, unsigned char *bcc)
{
    int size = 0;

    for (unsigned int i = 0; i < sizeBuffer; i++)
    {
        if (bcc != NULL)
            *bcc ^= frame[i];

        if (frame[i] == ESC_BYTE)

```

```

    {

        data[size++] = ESC_BYTE;

        data[size++] = BYTE_STUFFING_ESCAPE;

        break;

    }

    if (frame[i] == FLAG)

    {

        data[size++] = ESC_BYTE;

        data[size++] = BYTE_STUFFING_FLAG;

        break;

    }

    data[size++] = frame[i];

}

return size;

}

////////////////////

////////// TLV

////////////////////

/**



 * @brief reads next tlv

 *

 * @param buf

 * @param type

 * @param length

 * @param value

 * @return int

 */

```

```

int next_tlv(unsigned char *buf, unsigned char *type, unsigned char *length,
unsigned char **value)

{
    *type = buf[0];
    *length = buf[1];
    *value = buf + 2;
    return 2 + *length;
}

///////////////////////////////
// STATE MACHINE
///////////////////////////////

/** 
 * @brief state machine to handle the information of the frame and how to
behave
 *
 * @param trama
 * @param byte
 */
void state_machine_handler(Trama *trama, unsigned char byte)
{
    switch (trama->state)
    {
        case S_REJ:
        case S_END:
            trama->state = S_START;
        case S_START:
            if (byte == FLAG)

```

```

        trama->state = S_FLAG;

        break;

case S_FLAG:

    trama->data_size = 0;

    if (byte == A_SEND || byte == A_RESPONSE)

    {

        trama->state = S_ADR;

        trama->adr = byte;

        break;

    }

    if (byte == FLAG)

        break;

    trama->state = S_START;

    break;

case S_ADR:

    if (byte == C_SET || byte == C_DISC || byte == C_UA || byte ==
C_RR_(0) || byte == C_RR_(1) || byte == C_RJ_(0) || byte == C_RJ_(1) || byte
== C_DATA_(0) || byte == C_DATA_(1))

    {

        trama->state = S_CTRL;

        trama->ctrl = byte;

        trama->bcc = createBCC_header(trama->adr, trama->ctrl);

        break;

    }

    if (byte == FLAG)

    {

        trama->state = S_FLAG;

```

```

        break;

    }

    trama->state = S_START;
    break;

case S_CTRL:
    if (byte == trama->bcc)
    {
        trama->state = S_BCC1;
        break;
    }
    if (byte == FLAG)
    {
        trama->state = S_FLAG;
        break;
    }
    trama->state = S_START;
    break;

case S_BCC1:
    if (byte == FLAG)
    {
        if (trama->ctrl == C_DATA_0 || trama->ctrl == C_DATA_1)
        {
            trama->state = S_FLAG;
            break;
        }
        trama->state = S_END;
    }

```

```

        break;

    }

    if ((trama->ctrl == C_DATA_(0) || trama->ctrl == C_DATA_(1)) &&
trama->data != NULL)

    {

        trama->data_size = 0;

        if (byte == ESC_BYTEx)

        {

            trama->bcc = 0;

            trama->state = S_ESC;

            break;

        }

        trama->data[trama->data_size++] = byte;

        trama->bcc = byte;

        trama->state = S_DATA;

        break;

    }

    trama->state = S_START;

    break;

}

case S_BCC2:

    if (byte == 0)

    {

        trama->data[trama->data_size++] = trama->bcc;

        trama->bcc = 0;

        break;

    }

    if (byte == FLAG)

    {

```

```

        trama->state = S_END;

        break;

    }

    if (byte == ESC_BYTE)

    {

        trama->data[trama->data_size++] = trama->bcc;

        trama->bcc = 0;

        trama->state = S_ESC;

        break;

    }

    trama->data[trama->data_size++] = trama->bcc;

    trama->data[trama->data_size++] = byte;

    trama->bcc = byte;

    trama->state = S_DATA;

    break;

case S_DATA:

    if (byte == trama->bcc)

    {

        trama->state = S_BCC2;

        break;

    }

    if (byte == ESC_BYTE)

    {

        trama->state = S_ESC;

        break;

    }

    if (byte == FLAG)

    {

```

```

        trama->state = S_REJ;

        break;

    }

    trama->data[trama->data_size++] = byte;

    trama->bcc = createBCC_header(trama->bcc, byte);

    break;

case S_ESC:

    if (byte == BYTE_STUFFING_ESCAPE)

    {

        if (trama->bcc == ESC_BYTE)

        {

            trama->state = S_BCC2;

            break;

        }

        trama->bcc = createBCC_header(trama->bcc, ESC_BYTE);

        trama->data[trama->data_size++] = ESC_BYTE;

        trama->state = S_DATA;

        break;

    }

    if (byte == BYTE_STUFFING_FLAG)

    {

        if (trama->bcc == FLAG)

        {

            trama->state = S_BCC2;

            break;

        }

        trama->bcc = createBCC_header(trama->bcc, FLAG);

        trama->data[trama->data_size++] = FLAG;

```

```

        trama->state = S_DATA;

        break;
    }

    if (byte == FLAG)

    {

        trama->state = S_REJ;

        break;
    }

    trama->state = S_START;

    break;
}

```

macros.h:

```

#ifndef _MACROS_H_

#define _MACROS_H_


// tramas info

#define FLAG (0x7E)

#define A_SEND (0x03)

#define A_RESPONSE (0x01)


#define C_SET (0x03)

#define C_DISC (0x0B)

#define C_UA (0x07)

#define C_RR_(R) ((R) % 2 ? 0x85 : 0x05)

// #define C_RR_0 (0x05)

// #define C_RR_1 (0x85)

```

```
#define C_RJ_(J) ((J) % 2 ? 0x81 : 0x01)

//#define C_RJ_0 (0x01)
//#define C_RJ_1 (0x81)

#define C_DATA_(D) ((D) % 2 ? 0x40 : 0x00)

//#define C_DATA_0 (0x00)
//#define C_DATA_1 (0x40)

// App Layer

#define CTRL_DATA (1)

#define CTRL_START (2)

#define CTRL_END (3)

#define T_SIZE (0)

#define BUF_SIZE (1024)

#define ESC_BYTE (0x7D)

#define BYTE_STUFFING_ESCAPE (0x5D)

#define BYTE_STUFFING_FLAG (0x5E)

#endif
```