1. Introdução

Este trabalho realizado no âmbito da unidade curricular de Redes de Computadores do 3ºano da licenciatura em Engenharia Informática e Computação, da FEUP, tinha como objetivo implementar um protocolo de ligação de dados para a transmissão de um ficheiro entre dois computadores através da Porta Série RS-232.

No decorrer deste relatório, iremos explicar e esclarecer como desenvolvemos esse protocolo através de programação na linguagem C.

O projeto envolveu a criação de uma arquitetura com duas camadas principais, a camada de Ligação (Link Layer) e a camada de Aplicação (Application Layer). A camada de Ligação cuidou de todas as operações relacionadas à comunicação, desde o estabelecimento da conexão até o encerramento, incluindo a criação, envio e validação de pacotes entre os dois computadores. A camada de Aplicação interagiu com a camada de Ligação para a transmissão e receção de dados.

O protocolo de ligação lógica foi projetado para estabelecer uma conexão inicial entre os computadores, transmitir dados de forma confiável e encerrar a comunicação de maneira controlada. Para isso, foram implementadas funções como llopen() para estabelecer a conexão, llwrite() para enviar dados com verificação de erros e llclose() para encerrar a comunicação de forma segura. Tanto o transmissor quanto o receptor tiveram seus fluxos de chamadas de funções distintos para atender às suas funções específicas.

No geral, o projeto demonstrou a implementação bem-sucedida de um protocolo de ligação de dados para a transmissão de arquivos entre computadores usando a porta série RS-232, com a ênfase na confiabilidade e integridade dos dados durante a transmissão.

2. Arquitetura

Blocos Funcionais

No desenvolvimento do projeto, de forma a respeitar a independência entre camadas, foram utilizadas duas camadas principais: a Link Layer e a Application Layer.

A Link Layer, ou camada de ligação, é onde existe e são feitas todas as ligações, desde o estabelecimento da ligação, até ao seu encerramento, passando pela criação, envio e validação de todos os pacotes entre os dois computadores, através da porta série.

A Application Layer é responsável por interagir com a Link Layer para a transmissão e receção de dados. Ela chama a Link Layer para enviar dados e receber dados transmitidos.

Interface

A interface entre a Application Layer e a Link Layer desempenha um papel crucial na comunicação entre essas duas camadas. Ela consiste em quatro funções específicas, chamadas pela Application Layer, mas executadas pela Link Layer. Estas funções desempenham funções essenciais na transmissão de dados e na gestão da conexão:

llopen(): Esta função é responsável por estabelecer a ligação entre os dois computadores através do envio de tramas de controlo. Ela inicia o processo de comunicação e prepara o programa para a transmissão de dados.

llread(): A função llread() lê e verifica as tramas de dados recebidas. Além disso, ela envia a resposta apropriada caso a trama tenha sido recebida com erros ou não. É responsável por garantir a integridade dos dados durante a transmissão.

llwrite(): Esta função é encarregada de escrever e enviar as tramas de informação. Ela é responsável por transformar os dados da Application Layer em tramas de dados apropriadas para a transmissão pela Link Layer.

llclose(): A função llclose() é responsável por encerrar a conexão entre os dois computadores através da porta série. Ela realiza as ações necessárias para finalizar a comunicação de forma controlada e segura.

3. Estrutura do código

Application Layer

Para a implementação e uso desta camada não foram implementadas estruturas de dados. Estas são as funções implementadas:

```
unsigned char* getControlPacket(int c, int oc, unsigned char* buffer);
```

Chamada pela função "divide" para contruir o control packet.

Argumentos:

- c: argumento para o compo de controlo
- oc: parametro para colocar em L1 e L2
- buffer: L1 e L2

```
unsigned char* getDataPacket(int bytesRead, unsigned char* read);
```

Chamada pela função "divide" para contruir o data packet.

Argumentos:

- read: pointer com a informação a transferir
- bytesRead: tamanho dos dados

```
int divide(const char* filename);
```

Chamada pela Application Layer dividir o ficheiro e enviar os packets para a Link Layer

Argumentos:

- nome do ficheiro a ser enviado

```
unsigned char* packet_rewrite(unsigned char* packet, int size);
```

Função chamada pela Application Layer para remover os três primeiros elementos do packet (o campo de controle e dois campos que representam o número de octetos que a função "llread" lê).

Argumentos:

- packet: packet lido pela "llread"
- size: tamanho do packet

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

Função principal da Application Layer.

Argumentos:

- serialPort: nome da Serial port (ex: /dev/ttyS0)
- role: papel da Application layer ("tx" - transmissor ou "rx" - recetor)
- baudrate: baudrate da Serial port
- nTries: numero máximo de tentativas de envio de frames
- timeout: tempo de espera até um frame ser reenviado
- filename: nome do ficheiro a ser enviado/recebido

Link Layer

```
typedef enum {
    START_STATE,
    FLAG_STATE,
    A_STATE,
    C_STATE,
    BCC_STATE,
    DATA_STATE,
    ESC_STATE,
    END_STATE
} State;
```

```
typedef struct
{
    char serialPort[50];
    LinkLayerRole role;
    int baudRate;
    int nRetransmissions;
    int timeout;
} LinkLayer;
```

```
typedef enum
{
    LlTx,
    LlRx,
} LinkLayerRole;
```

Para uso desta camada houve a necessidade de utilizar três estruturas de dados, LinkLayer, na qual obtemos informação sobre todos os dados necessários para a transferência, LinkLayerRole, onde identificamos o transmissor e o recetor, e por fim State, onde identificamos o estado relativo à leitura dos dados das tramas de informação recebidas.

Funções implementadas:

```
void alarmHandler(int signal);
```

Função para a criação do alarme

```
int llopen(LinkLayer connectionParameters);
```

Estabelece a connecção

```
unsigned char state_write();
```

State machine usada na função "llwrite" para validar os frames

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

Envia a informação do "buf" que tem tamanho "bufSize"

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

Lê a informação do packet

```
int llclose(int showStatistics);
```

Fecha a connecção e imprime as estatísticas se showStatistics for TRUE.

4. Casos de uso principais

Derivado do facto de haver duas possibilidades de papeis quando se executa o programa, recetor e transmissor, e as suas funções não serem as mesmas, tivemos de diferenciar a sequencia de chamada de funções por parte de cada um.

Transmissor:

- Llopen() -> utilizada para testar a disponibilidade e conexão entre os dois computadores, pela troca de pacotes de controlo e supervisão.
- Divide() -> onde vai ser feita a abertura e parcelamento do ficheiro para ser enviado.
- getControlPacket() -> construção de um pacote de controlo para informar o recetor que a transferência de dados vai iniciar e qual o tamanho que vai ser transmitido.
- getDataPacket() -> devolve uma trama com os dados que vão ser transferidos e o tamanho desta trama.
- Llwrite() -> Cria uma nova trama a partir da recebida com uma Head de controlo, com Flag, A, C, BCC1, faz stuff dos dados recebidos na trama em parâmetro e insere na trama a enviar e junta um BCC2 relativo ao xor entre todos os dados e uma Flag final.
- State_write() -> state machine do transmissor, onde é feita a leitura de todos os elementos recebidos e valida estes elementos através de uma state Machine.
- Llclose() -> função para dar termino a ligação entre os dois computadores, onde é enviada e lida uma trama de controlo.

Recetor:

- Llopen() -> utilizada para testar a disponibilidade e conexão entre os dois computadores, pela troca de pacotes de controlo e supervisão.

- Llread() -> função que le os dados enviados pelo transmissor, valida através de uma state machine e campos de controlo (BCC1 e BCC2), após destuffing, e envia tramas de rejeição ou aceitação mediante a validação dos dados.
- Packet_rewrite() -> função acessória para remover os elementos de controlo da trama lida de modo a ficar apenas com dados para serem escritos no ficheiro.
- Llcose() -> função para dar termino a ligação entre os dois computadores, onde é enviada e lida uma trama de controlo.

5. Protocolo de Ligação Lógica

Esta é a camada de mais baixo nível que tem por encargo, conectar, enviar, receber e fechar a ligação.

De modo a estabelecermos uma ligação inicial entre os dois pontos da porta utilizamos a função llopen(), onde o emissor envia uma trama de controlo inicial (SET) que o recetor deve ler e validar pela maquina de estados, enviando como confirmação uma trama UA, que deve também ser lida e validada do lado do transmissor. Apenas após o envio e confirmação das duas tramas a ligação foi corretamente estabelecida, podendo então passar ao próximo passo do protocolo.

Estando neste momento no llwrite(), a primeira trama a ser enviada é uma trama com um campo de controlo, sendo isso possível de ser observado na cabeça da trama pelo seu valor inicial de 2 que informa que vai iniciar a transferência de dados com e o tamanho dos dados que vai transmitir. Posteriormente esta função recebe tramas de dados, procedendo primeiro à sua codificação através do mecanismo de stuffing de modo a evitar conflitos e possíveis más interpretações daquilo que são dados ou flags. Também são inseridas na trama uma flag inicial, um campo A, um campo C que deve ser respeitado e alternado de modo a ter garantias que a tramas estão a ser bem enviadas e lidas e não há problemas na conexão e também um BCC1 e BCC2, sendo o primeiro relativo aos dados iniciais de A e C e o segundo relativo a toda a informação a ser enviada, antes de ser efetuado o stuffing. Com a trama devidamente construída passamos então ao envio da mesma. Esta trama é enviada e em casa de não obtenção de resposta (no tempo anteriormente estipulado) é enviada novamente, até um máximo de 3 envios, no caso de haver uma resposta negativa que simboliza alguma falha na transmissão (REJ0 ou REJ1) a trama é novamente enviada sem ser contabilizado como envio.

A leitura desta trama é efetuado pelo llread(), que enquanto passa a trama pela sua maquina de estados de modo a avaliar a sua correta receção vai procedendo ao seu destuffing. Apenas no final, depois de verificar que a trama foi bem recebida faz o xor de todos os elementos de dados para verificar com o BCC2. Se tudo isto estiver correto, então podemos afirmar que a trama foi corretamente recebida e enviar uma trama de aceitação (RR0 ou RR1). No caso de alguma destas incidências não se verificar, é enviada uma trama de rejeição (REJ0 ou REJ1).

De modo a efetuar o término da ligação o emissor passa para a função llclose(), onde ira inicialmente enviar uma trama de controlo para o recetor com um campo inicial de C igual a 3, informando que o envio de dados foi terminado. O recetor deve ler e processar o DISC e enviar um DISC novamente, que é lido e aceite pelo transmissor, que envia um UA e dá por encerrada a

ligação. Se durante o envio dos dados ocorrer um TIMEOUT ou for esgotado o numero de tentativas de envio, o emissor passa diretamente para a llclose().

6. Protocolo de aplicação

Sendo esta a camada mais próxima do utilizador, é nela que se define qual o ficheiro que vai ser transmitido, bem como a porta série, o número de bytes máximos de cada trama, numero máximo de transmissões e o tempo que deve ser aguardado entre transmissões. A transferência dos dados ira ser feita através da utilização da API da LinkLayer, onde é feita a transferência dos dados.

A primeira coisa a ser feita pela aplicação é verificar que os dois computadores estão ligados e disponíveis para comunicar, utilizando para isso a llopen(). No caso de esta função dar um retorno positivo, o recetor é imediatamente encaminhado para a função llread(), enquanto que o transmissor passa as suas funções para a divide(). Nesta função, é inicialmente construído um pacote de controlo, com informação de que se vai iniciar o envio dos dados e qual o tamanho total dos dados, com a ajuda da getControlPacket().

O recetor lê esta trama e passa para um ciclo de leitura de tramas de informação.

Após o envio do primeiro pacote de controlo é iniciada, verdadeiramente, a transferência dos dados. O ficheiro á aberto e em cada iteração são lidos o máximo de bytes permitidos para transferência menos 3, para que na função getDataPacket(), se possa juntar o campo de controlo a informar que a trama é de dados e L1 e L2, representado o numero de octetos a ser enviado. Depois de construída esta trama, a mesma é enviada para a llwrite(), para que comunique com o recetor. Enquanto houver dados para ler do ficheiro esta sequencia mantem-se.

No lado do recetor, a leitura das tramas de dados é feita de um modo continuo e sempre inserindo estes dados num ficheiro, de cada vez que a função llread() tem um retorno positivo. Por parte do recetor é feita uma verificação relativa ao elemento na posição 0 da trama recebida, uma vez que se esse elemento for diferente de um 1, essa trama não deve ser colocada dentro do ficheiro.

Esta verificação é feita, uma vez que no final do envio de todos os dados por parte do transmissor, é construída uma trama de controlo, com a ajuda da função getControlPacket, com o valor de C igual a 3, informando que a trama é de finalização que as informações dentro desta trama não são dados do ficheiro, dando por terminado o envio do ficheiro. Passa então para a função llclose().

O recetor ao ler esta trama, não efetua a inserção da mesma no ficheiro. Dá assim por terminada a receção de dados a inserir no ficheiro, fecha o mesmo e passa para o llclose(), onde deve passar pela sua maquina de estados verificando que recebeu um DISC, e em caso positivo envia um DISC também de confirmação. Se tudo estiver correto entre os dois computadores, o emissor deve fechar apenas depois de enviar um UA, o qual é recebido pelo recetor e proceder então ao fecho da comunicação.

7. Validação

Para garantir a correta implementação do protocolo desenvolvido, foram efetuados os seguintes testes:

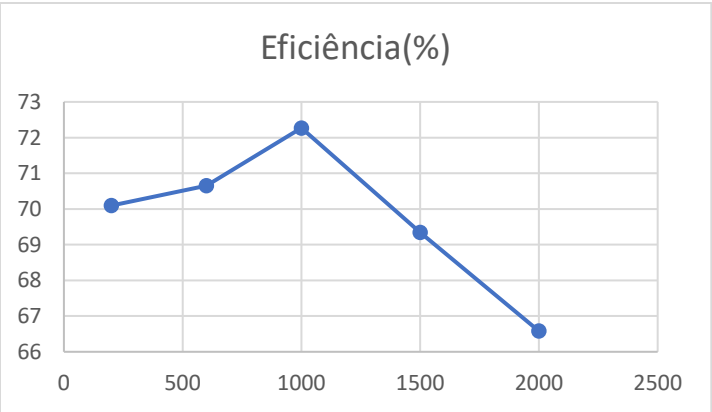
- Transferência de ficheiros de diferentes tamanhos
- Utilização de diferentes *baudrates*
- Transferência de packets de *tamanhos* diferentes
- Interrupções da Porta Série
- Introdução de ruído durante a transferência

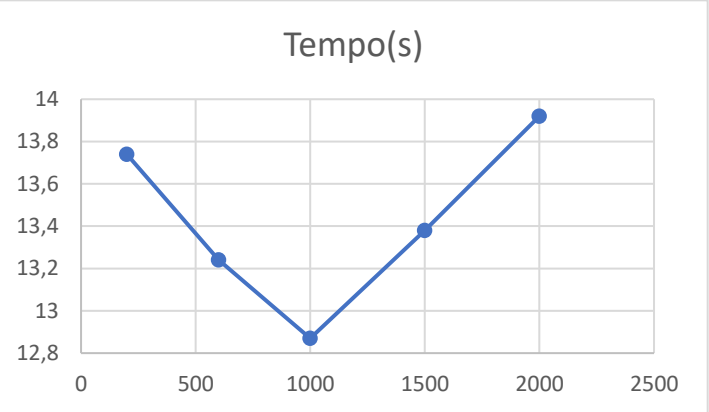
Durante estes testes o ficheiro foi enviado e recebido na totalidade. Foi introduzido por nós um output para a eventualidade de o programa detetar erros, tanto no recetor como no transmissor. Durante a execução destes testes foram várias as vezes em que vimos esse output ser impresso na consola, não tendo sido isso impedimento de envio da mensagem de maneira correta.

8. Eficiência do protocolo de ligação de dados

Variação do tamanho máximo da trama

MAX_PAYLOAD_SIZE	BAUDRATE	Tempo(s)	Eficiência(%)
1000	9600	12,87	72,27
1000	19200	12,91	36,04
1000	38400	11,7	19,87
1000	76800	11,71	9,94
1000	115200	11,7	6,62

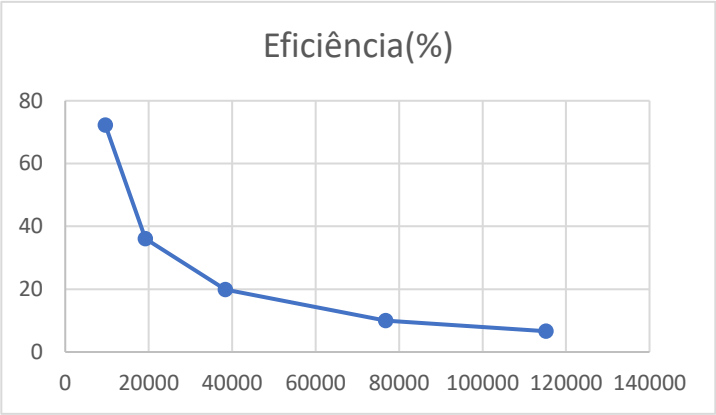


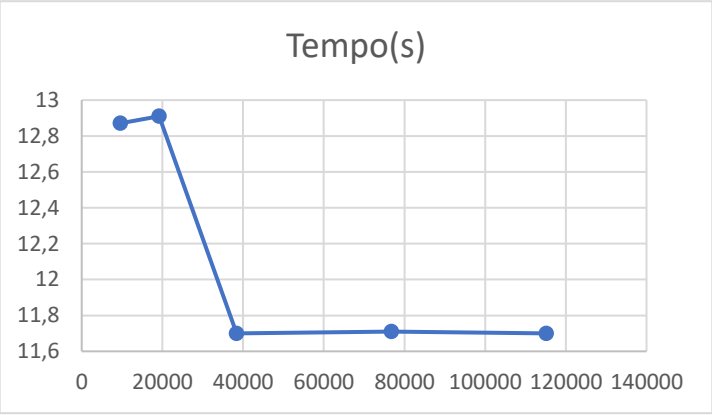


Com a variação do tamanho máximo da não existem variações significativas, o que não vai de encontro aqueles que eram os resultados esperados. Era expectável que com o aumentar do tamanho máximo da trama, o tempo diminuísse, no entanto, vemos que apesar de haver uma ligeira diminuição quando o tamanho máximo da trama se fixa nos 1000, a variação não é significativa.

Variação do tamanho da Baudrate

MAX_PAYLOAD_SIZE	BAUDRATE	Tempo(s)	Eficiência(%)
1000	9600	12,87	72,27
1000	19200	12,91	36,04
1000	38400	11,7	19,87
1000	76800	11,71	9,94
1000	115200	11,7	6,62





Quando fizemos variar a Baudrate conseguimos perceber que existem algumas diferenças, apesar de não serem extremamente muito siginificantes. Apesar de existirem melhorias no tempo de execução e transferência do programa, essa diferença excede em muito pouco um segundo, o que não é algo muito relevante. De salientar que os tempos estão divididos em dois grupos, um com um tamanho de menos de 19200, inclusive, e outro grupo quando a Baudrate excede este valor.

9. Conclusões

O projeto desenvolveu com sucesso um protocolo de ligação de dados para a transmissão de ficheiros entre computadores utilizando a porta série RS-232. A arquitetura do protocolo incluiu duas camadas principais, a camada de Ligação (Link Layer) e a camada de Aplicação (Application Layer), com funções bem definidas para estabelecer a conexão, transmitir dados com verificação de erros e encerrar a comunicação de forma controlada. A implementação foi testada em diversos cenários, garantindo a integridade e confiabilidade dos dados transmitidos.

ANEXOS

application_layer.h

```
 1. // Application layer protocol header.
 2. // NOTE: This file must not be changed.
 3.
 4. #ifndef _APPLICATION_LAYER_H_
 5. #define _APPLICATION_LAYER_H_
 6. #define MAX_PAYLOAD_SIZE 1000
 7.
 8.
 9.
10. // Function called by divide to build a control packet
11. // Arguments:
12. //          c: argument to put in control field
13. //          oc: parameter to put in L1 and L2
14. //          buffer: L1 and L2
15. unsigned char* getControlPacket(int c, int oc, unsigned char* buffer);
16.
17.
18.
19. // function called by the divide to build the data packet to send
20. // Arguments:
21. //           read: pointer with the data to transfer
22. //           bytesRead: size of bytes read saved in the pointer read
23. unsigned char* getDataPacket(int bytesRead, unsigned char* read);
24.
25.
26.
27.
28. // Function called by the Application Layer  to divide and send the packet to the llwrite
29. // Arguments: filename: name of tthe file to be opened and sent to the receiver
30. // returns -1 on error and 1 on success
31. //
32. int divide(const char* filename);
33.
34.
35. // Function called by the Application layer to remove the first three elements of the
packet, which are the control field, and two fields that are the number of octets that llread
function read
36. // Arguments: packet: packet read by the llread
37. //            size: size of the packet
38. // returns a new unsigned cha* with all the elements except the first three
39. unsigned char* packet_rewrite(unsigned char* packet, int size);
40.
41. // Application layer main function.
42. // Arguments:
43. //   serialPort: Serial port name (e.g., /dev/ttyS0).
44. //   role: Application role {"tx", "rx"}.
45. //   baudrate: Baudrate of the serial port.
46. //   nTries: Maximum number of frame retries.
47. //   timeout: Frame timeout.
48. //   filename: Name of the file to send / receive.
49. void applicationLayer(const char *serialPort, const char *role, int baudRate,
50.                       int nTries, int timeout, const char *filename);
51.
52. #endif // _APPLICATION_LAYER_H_
53.
```

application_layer.c

```c
// Link layer protocol implementation

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

// MISC
#define _POSIX_SOURCE 1 // POSIX compliant source


#define FLAG 0x7E
#define DISC 0x0B
#define ESC 0x7D
#define IN0 0x00
#define IN1 0x40
#define RR0 0x05
#define RR1 0x85
#define REJ0 0x01
#define REJ1 0x81
#define TRUE 1
#define FALSE 0

int alarmCount = 0;
int alarmEnabled = FALSE;
int STOP = FALSE;
int flagFrame = 0;
int timeout=0;
int nRetransmissions=0;
int fd=-1;
struct termios oldtio;
int ttt=0;
LinkLayerRole _role;


void alarmHandler(int signal)
{
    alarmEnabled = FALSE;
    alarmCount++;

    printf("Alarm #%d\n", alarmCount);
}




////////////////////////////////////////////////
// LLOPEN
////////////////////////////////////////////////
int llopen(LinkLayer connectionParameters)
{
    fd = open(connectionParameters.serialPort, O_RDWR | O_NOCTTY);

```

```
61.     if (fd < 0)
62.     {
63.         perror(connectionParameters.serialPort);
64.         exit(-1);
65.     }
66.
67.     struct termios newtio;
68.
69.     if (tcgetattr(fd, &oldtio) == -1)
70.     {
71.         perror("tcgetattr");
72.         exit(-1);
73.     }
74.
75.     memset(&newtio, 0, sizeof(newtio));
76.
77.     newtio.c_cflag = connectionParameters.baudRate | CS8 | CLOCAL | CREAD;
78.     newtio.c_iflag = IGNPAR;
79.     newtio.c_oflag = 0;
80.
81.     newtio.c_lflag = 0;
82.     newtio.c_cc[VTIME] = connectionParameters.timeout*10; // Inter-character timer unused
83.     newtio.c_cc[VMIN] = 0;  // Blocking read until 5 chars received
84.
85.
86.     tcflush(fd, TCIOFLUSH);
87.
88.     if (tcsetattr(fd, TCSANOW, &newtio) == -1)
89.     {
90.         perror("tcsetattr");
91.         exit(-1);
92.     }
93.
94.
95.
96.     _role = connectionParameters.role;
97.
98.     timeout = connectionParameters.timeout;
99.     nRetransmissions = connectionParameters.nRetransmissions;
100.    int cont=0;
101.
102.    unsigned char buf_w[5] = {FLAG, 0x03, 0x03, 0x03^0x03, FLAG};
103.    unsigned char buf_r[5] = {FLAG, 0x03, 0x07, 0x03^0x07, FLAG};
104.    unsigned char c;
105.
106.
107.    if (_role == LlTx)
108.    {
109.        (void)signal(SIGALRM, alarmHandler);
110.
111.        while (cont != connectionParameters.nRetransmissions)
112.        {
113.            enum State state = START_STATE;
114.
115.            if (alarmEnabled == FALSE)
116.            {
117.                int bytes = write(fd, buf_w, 5);
118.                alarm(timeout);
119.                alarmEnabled = TRUE;
120.            }
121.            while (read(fd, &c, 1)>0){
122.                switch (state) {
123.                    case START_STATE:
124.                        if (c==FLAG) state= FLAG_STATE;
125.                        break;
```

```
                    case FLAG_STATE:
                        if (c==0x03) state=A_STATE;
                        else if (c==FLAG) state= FLAG_STATE;
                        else state= START_STATE;
                        break;
                    case A_STATE:
                        if (c==0x07) state = C_STATE;
                        else if (c==FLAG) state= FLAG_STATE;
                        else state= START_STATE;
                        break;
                    case C_STATE:
                        if (c==(0x03^0x07)) state=BCC_STATE;
                        else if (c==FLAG) state= FLAG_STATE;
                        else state= START_STATE;
                        break;
                    case BCC_STATE:
                        if (c==FLAG) return 1;
                        else state= START_STATE;
                        break;


                }
            }
            alarmEnabled=FALSE;
            cont++;
        }
        if (connectionParameters.nRetransmissions==cont){
            printf("UNABLE TO ESTABLISH CONNECTION\n");
            if (tcsetattr(fd, TCSANOW, &oldtio) == -1)
            {
                perror("tcsetattr");
                exit(-1);
            }
            close(fd);
            return -1;
        }


        alarmEnabled=FALSE;
        alarmCount=0;
        return 1;
    }
    else if (_role == LlRx)
    {

        unsigned char c;
        enum State state=START_STATE;
        while (state != END_STATE){
            read(fd, &c, 1);
            switch (state) {
                case START_STATE:
                    if (c==FLAG) state= FLAG_STATE;
                    break;
                case FLAG_STATE:
                    if (c==0x03) state=A_STATE;
                    else if (c==FLAG) state= FLAG_STATE;
                    else state= START_STATE;
                    break;
                case A_STATE:
                    if (c==0x03) state = C_STATE;
                    else if (c==FLAG) state= FLAG_STATE;
                    else state= START_STATE;
                    break;
                case C_STATE:
                    if (c==(0x03^0x03)) state=BCC_STATE;
```

```
                    else if (c==FLAG) state= FLAG_STATE;
                    else state= START_STATE;
                    break;
                case BCC_STATE:
                    if (c==FLAG) {
                        state=END_STATE;
                        write(fd, buf_r, 5);
                    }
                    else state= START_STATE;
                    break;
                case END_STATE:
                    return 1;

            }
        }



        return 1;
    }

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


    printf("error");
    return -1;

}



unsigned char state_write(){
    enum State state = START_STATE;
    unsigned char read2, save;

    while (state!=END_STATE){
        if(read(fd, &read2, 1)>0){
            switch (state) {
                case START_STATE:
                    if (read2==FLAG)state=FLAG_STATE;
                    break;
                case FLAG_STATE:
                    if (read2==0x03) state=A_STATE;
                    else if (read2==FLAG) state=FLAG_STATE;
                    else state=START_STATE;
                    break;
                case A_STATE:
                    if (read2==RR0){
                        flagFrame=0;
                        save=read2;
                        state=C_STATE;
                    }
                    else if(read2==RR1){
                        flagFrame=1;
                        save=read2;
                        state=C_STATE;
                    }
                    else if(read2==REJ0){
                        flagFrame=0;
                        save=read2;
                        state=C_STATE;
```

```
                    }
                    else if (read2==REJ1){
                        flagFrame=1;
                        save=read2;
                        state=C_STATE;
                    }
                    else if (read2==DISC) save=read2;
                    else if (read2==FLAG) state=FLAG_STATE;
                    else state=START_STATE;
                    break;
                case C_STATE:
                    if (read2==(0x03^save))state= BCC_STATE;
                    else if (read2==FLAG) state=FLAG_STATE;
                    else state=START_STATE;
                    break;
                case BCC_STATE:
                    if (read2==FLAG)state = END_STATE;
                    else state=START_STATE;
                    break;
                default:
                    break;
            }

        }
        else break;
    }

    return save;

}




////////////////////////////////////////////////
// LLWRITE
////////////////////////////////////////////////
int llwrite(const unsigned char *buf, int bufSize)
{

    (void)signal(SIGALRM, alarmHandler);

    // TODO

    if (fd < 0) {
        perror("Invalid file descriptor");
        return -1;
    }

    unsigned char* stuffed = (unsigned char*)malloc(2 * bufSize * sizeof(unsigned char));
    stuffed[0]=FLAG;
    stuffed[1]=0x03;
    if (flagFrame==0){
        stuffed[2]=IN0;
    }
    else if (flagFrame==1){
        stuffed[2]=IN1;
    }
    stuffed[3]=stuffed[1]^stuffed[2];
    int cont=4;
    unsigned char xor=0;
    for(int i=0; i<bufSize; i++){
        xor^=buf[i];
        if (buf[i]==0x7e){
```

```
            stuffed[cont]=ESC;
            stuffed[cont+1]=0x5e;
            cont+=2;
        }
        else if (buf[i]==0x7d){
            stuffed[cont]=ESC;
            stuffed[cont+1]=0x5d;
            cont+=2;
        }
        else{
            stuffed[cont]=buf[i];
            cont++;
        }
    }


    stuffed[cont]=xor;
    stuffed[cont+1]=FLAG;

    cont+=2;

    int trans = nRetransmissions;


    int bytes=0;
    alarmEnabled=FALSE;
    alarmCount=0;


    while (trans!=0){
        if (alarmEnabled == FALSE)
        {
            bytes = write(fd, stuffed, cont);
            alarm(timeout);
            alarmEnabled = TRUE;
        }
        if (bytes != cont) {
            fd=-1;
            perror("Error writing to serial port");
            return -1;
        }
        unsigned char read = state_write();

        if (read==RR1 || read==RR0){
            alarm(0);
            break;
        }
        else if (read==REJ0 || read==REJ1){
            alarm(0);
        }
        alarmEnabled=FALSE;
        trans--;

    }


    if (trans==0){
        return -1;
    }

   return bytes;


}

```

```
386. ////////////////////////////////////////////////
387. // LLREAD
388. ////////////////////////////////////////////////
389. int llread(unsigned char* packet)
390. {
391. 
392.     unsigned char bytes;
393.     enum State state = START_STATE;
394.     unsigned char c_bye;
395.     int i=0;
396. 
397. 
398.     while (state!=END_STATE){
399.         if (read(fd, &bytes, 1)<0){
400.             return 1;
401.         }
402.         switch (state) {
403.             case START_STATE:
404.                 if (bytes == FLAG) state=FLAG_STATE;
405.                 break;
406.             case FLAG_STATE:
407.                 if (bytes==0x03) state=A_STATE;
408.                 else if (bytes==FLAG) state=FLAG_STATE;
409.                 else state=START_STATE;
410.                 break;
411.             case A_STATE:
412.                 if (bytes==IN0 || bytes==IN1){
413.                     state = C_STATE;
414.                     c_bye = bytes;
415.                     if (bytes==IN0) flagFrame=0;
416.                     else flagFrame=1;
417.                 }
418.                 else if (bytes==FLAG) state=FLAG_STATE;
419.                 else state=START_STATE;
420.                 break;
421.             case C_STATE:
422.                 if (bytes== (0x03^c_bye))state=DATA_STATE;
423.                 else if (bytes==FLAG) state=FLAG_STATE;
424.                 else state=START_STATE;
425.                 break;
426.             case DATA_STATE:
427.                 if (bytes==FLAG){
428.                     unsigned char BCC2 = packet[i-1];
429.                     unsigned char xor=0;
430.                     for (int j=0; j<i-1; j++){
431.                         xor^=packet[j];
432.                     }
433. 
434.                     if (xor==BCC2) {
435.                         state = END_STATE;
436.                         if (flagFrame == 0) {
437.                             unsigned char send_RR1[5] = {FLAG, 0x03, RR1, 0x03 ^ RR1,
FLAG};
438.                             write(fd, send_RR1, 5);
439.                             return i-1;
440.                         }
441.                         else if (flagFrame==1){
442.                             unsigned char send_RR0[5] = {FLAG, 0x03, RR0, 0x03 ^ RR0,
FLAG};
443.                             write(fd, send_RR0, 5);
444.                             return i-1;
445.                         }
446.                     }
447.                     else{
448.                         printf("REJECTED\n");
```

```
449.                         if (flagFrame == 0) {
450.                             unsigned char send_REJ0[5] = {FLAG, 0x03, REJ0, 0x03 ^ REJ0,
FLAG};
451.                             write(fd, send_REJ0, 5);
452.                             return -1;
453.                         } else if (flagFrame == 1) {
454.                             unsigned char send_REJ1[5] = {FLAG, 0x03, REJ1, 0x03 ^ REJ1,
FLAG};
455.                             write(fd, send_REJ1, 5);
456.                             return -1;
457.                         }
458.                     }
459.                 }
460.                 else if (bytes==ESC) state = ESC_STATE;
461.                 else packet[i++]=bytes;
462.                 break;
463.             case ESC_STATE:
464.                 state=DATA_STATE;
465.                 if (bytes==0x5e) packet[i++]=0x7e;
466.                 else if (bytes==0x5d) packet[i++]=0x7d;
467.                 else{
468.                     packet[i++]=ESC;
469.                     packet[i++]=bytes;
470.                 }
471.                 break;
472.
473.         }
474.
475.     }
476.
477.     return -1;
478.
479. }
480.
481. ////////////////////////////////////////////////
482. // LLCLOSE
483. ////////////////////////////////////////////////
484. int llclose(int showStatistics)
485. {
486.
487.     (void)signal(SIGALRM, alarmHandler);
488.
489.     alarmEnabled=FALSE;
490.     alarmCount=0;
491.     unsigned char buf_disc[5] = {FLAG, 0x03, 0x0B, 0x03^0x0B, FLAG};
492.     unsigned char buf_ua[5] = {FLAG, 0x03, 0x07, 0x03^0x07, FLAG};
493.     unsigned char rec;
494.     int cont=0;
495.     unsigned char c;
496.
497.
498.     if (_role==LlTx) {
499.         (void) signal(SIGALRM, alarmHandler);
500.         enum State state = START_STATE;
501.
502.
503.         while (cont != nRetransmissions && state != END_STATE) {
504.
505.             if (alarmEnabled == FALSE) {
506.                 int bytes = write(fd, buf_disc, 5);
507.                 alarm(timeout);
508.                 alarmEnabled = TRUE;
509.             }
510.             while (state != END_STATE) {
511.                 read(fd, &c, 1);
```

```
512.                 switch (state) {
513.                     case START_STATE:
514.                         if (c == FLAG) state = FLAG_STATE;
515.                         break;
516.                     case FLAG_STATE:
517.                         if (c == 0x01) state = A_STATE;
518.                         else if (c == FLAG) state = FLAG_STATE;
519.                         else state = START_STATE;
520.                         break;
521.                     case A_STATE:
522.                         if (c == DISC) state = C_STATE;
523.                         else if (c == FLAG) state = FLAG_STATE;
524.                         else state = START_STATE;
525.                         break;
526.                     case C_STATE:
527.                         if (c == (0x01 ^ 0x0B)) state = BCC_STATE;
528.                         else if (c == FLAG) state = FLAG_STATE;
529.                         else state = START_STATE;
530.                         break;
531.                     case BCC_STATE:
532.                         if (c == FLAG) {
533.                             state = END_STATE;
534.                             alarm(0);
535.                         } else state = START_STATE;
536.                         break;
537.
538.
539.                 }
540.             }
541.             alarmEnabled = FALSE;
542.             cont++;
543.         }
544.
545.         write(fd, buf_ua, 5);
546.     }
547.
548.     else if (_role==LlRx) {
549.         unsigned char bytes;
550.         enum State state_r = START_STATE;
551.         unsigned char c_bye;
552.         int i = 0;
553.
554.         unsigned char sv;
555.
556.         while (state_r != END_STATE) {
557.
558.             read(fd, &bytes, 1);
559.
560.             switch (state_r) {
561.                 case START_STATE:
562.                     if (bytes == FLAG) state_r = FLAG_STATE;
563.                     break;
564.                 case FLAG_STATE:
565.                     if (bytes == 0x03) state_r = A_STATE;
566.                     else if (bytes == FLAG) state_r = FLAG_STATE;
567.                     else state_r = START_STATE;
568.                     break;
569.                 case A_STATE:
570.                     if (bytes == DISC) {
571.                         sv=bytes;
572.                         state_r=C_STATE;
573.                     } else if (bytes == FLAG) state_r = FLAG_STATE;
574.                     else if (bytes==0x07){
575.                         state_r=C_STATE;
576.                         sv=bytes;
```

```
                    }
                    else state_r = START_STATE;
                    break;
                case C_STATE:
                    if (bytes== (0x03^sv))state_r=BCC_STATE;
                    else state_r=START_STATE;
                    break;
                case BCC_STATE:
                    if (bytes==FLAG){
                        if (sv==DISC){
                            unsigned char send_disc[5] = {FLAG, 0x01, DISC, 0x01 ^ DISC,
FLAG};
                            write(fd, send_disc, 5);
                            state_r = START_STATE;
                        }
                        else if (sv==0x07){
                            state_r=END_STATE;
                        }
                    }
                    break;

            }
        }

    }

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

}

```

link_layer.h

```c
// Link layer header.
// NOTE: This file must not be changed.

#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;

enum State{
    START_STATE,
    FLAG_STATE,
    A_STATE,
    C_STATE,
    BCC_STATE,
    DATA_STATE,
    ESC_STATE,
    END_STATE
};

// 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

void alarmHandler(int signal);
// Open a connection using the "port" parameters defined in struct linkLayer.
// Return "1" on success or "-1" on error.
int llopen(LinkLayer connectionParameters);

// State machine for the llwrite function
// returns the C element of the head of data packet
unsigned char state_write();

// Send data in buf with size bufSize.
// Return number of chars written, or "-1" on error.
int llwrite(const unsigned char *buf, int bufSize);

// 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);

```

```
63. #endif // _LINK_LAYER_H_
64.
```

link_layer.c

```
 1. // Link layer protocol implementation
 2.
 3. #include "link_layer.h"
 4. #include <fcntl.h>
 5. #include <stdio.h>
 6. #include <stdlib.h>
 7. #include <string.h>
 8. #include <sys/types.h>
 9. #include <sys/stat.h>
10. #include <termios.h>
11. #include <unistd.h>
12. #include <signal.h>
13.
14. // MISC
15. #define _POSIX_SOURCE 1 // POSIX compliant source
16.
17.
18. #define FLAG 0x7E
19. #define DISC 0x0B
20. #define ESC 0x7D
21. #define IN0 0x00
22. #define IN1 0x40
23. #define RR0 0x05
24. #define RR1 0x85
25. #define REJ0 0x01
26. #define REJ1 0x81
27. #define TRUE 1
28. #define FALSE 0
29.
30. int alarmCount = 0;
31. int alarmEnabled = FALSE;
32. int STOP = FALSE;
33. int flagFrame = 0;
34. int timeout=0;
35. int nRetransmissions=0;
36. int fd=-1;
37. struct termios oldtio;
38. int ttt=0;
39. LinkLayerRole _role;
40.
41.
42. void alarmHandler(int signal)
43. {
44.     alarmEnabled = FALSE;
45.     alarmCount++;
46.
47.     printf("Alarm #%d\n", alarmCount);
48. }
49.
50.
51.
52.
53.
54. ////////////////////////////////////////////////
55. // LLOPEN
```

```
56. ////////////////////////////////////////////////
57. int llopen(LinkLayer connectionParameters)
58. {
59.     fd = open(connectionParameters.serialPort, O_RDWR | O_NOCTTY);
60.
61.     if (fd < 0)
62.     {
63.         perror(connectionParameters.serialPort);
64.         exit(-1);
65.     }
66.
67.     struct termios newtio;
68.
69.     if (tcgetattr(fd, &oldtio) == -1)
70.     {
71.         perror("tcgetattr");
72.         exit(-1);
73.     }
74.
75.     memset(&newtio, 0, sizeof(newtio));
76.
77.     newtio.c_cflag = connectionParameters.baudRate | CS8 | CLOCAL | CREAD;
78.     newtio.c_iflag = IGNPAR;
79.     newtio.c_oflag = 0;
80.
81.     newtio.c_lflag = 0;
82.     newtio.c_cc[VTIME] = connectionParameters.timeout*10; // Inter-character timer unused
83.     newtio.c_cc[VMIN] = 0;  // Blocking read until 5 chars received
84.
85.
86.     tcflush(fd, TCIOFLUSH);
87.
88.     if (tcsetattr(fd, TCSANOW, &newtio) == -1)
89.     {
90.         perror("tcsetattr");
91.         exit(-1);
92.     }
93.
94.
95.
96.     _role = connectionParameters.role;
97.
98.     timeout = connectionParameters.timeout;
99.     nRetransmissions = connectionParameters.nRetransmissions;
100.    int cont=0;
101.
102.    unsigned char buf_w[5] = {FLAG, 0x03, 0x03, 0x03^0x03, FLAG};
103.    unsigned char buf_r[5] = {FLAG, 0x03, 0x07, 0x03^0x07, FLAG};
104.    unsigned char c;
105.
106.
107.    if (_role == LlTx)
108.    {
109.        (void)signal(SIGALRM, alarmHandler);
110.
111.        while (cont != connectionParameters.nRetransmissions)
112.        {
113.            enum State state = START_STATE;
114.
115.            if (alarmEnabled == FALSE)
116.            {
117.                int bytes = write(fd, buf_w, 5);
118.                alarm(timeout);
119.                alarmEnabled = TRUE;
120.            }
```

```
            while (read(fd, &c, 1)>0){
                switch (state) {
                    case START_STATE:
                        if (c==FLAG) state= FLAG_STATE;
                        break;
                    case FLAG_STATE:
                        if (c==0x03) state=A_STATE;
                        else if (c==FLAG) state= FLAG_STATE;
                        else state= START_STATE;
                        break;
                    case A_STATE:
                        if (c==0x07) state = C_STATE;
                        else if (c==FLAG) state= FLAG_STATE;
                        else state= START_STATE;
                        break;
                    case C_STATE:
                        if (c==(0x03^0x07)) state=BCC_STATE;
                        else if (c==FLAG) state= FLAG_STATE;
                        else state= START_STATE;
                        break;
                    case BCC_STATE:
                        if (c==FLAG) return 1;
                        else state= START_STATE;
                        break;


                }
            }
            alarmEnabled=FALSE;
            cont++;
        }
        if (connectionParameters.nRetransmissions==cont){
            printf("UNABLE TO ESTABLISH CONNECTION\n");
            if (tcsetattr(fd, TCSANOW, &oldtio) == -1)
            {
                perror("tcsetattr");
                exit(-1);
            }
            close(fd);
            return -1;
        }


        alarmEnabled=FALSE;
        alarmCount=0;
        return 1;
    }
    else if (_role == LlRx)
    {

        unsigned char c;
        enum State state=START_STATE;
        while (state != END_STATE){
            read(fd, &c, 1);
            switch (state) {
                case START_STATE:
                    if (c==FLAG) state= FLAG_STATE;
                    break;
                case FLAG_STATE:
                    if (c==0x03) state=A_STATE;
                    else if (c==FLAG) state= FLAG_STATE;
                    else state= START_STATE;
                    break;
                case A_STATE:
                    if (c==0x03) state = C_STATE;
```

```
                    else if (c==FLAG) state= FLAG_STATE;
                    else state= START_STATE;
                    break;
                case C_STATE:
                    if (c==(0x03^0x03)) state=BCC_STATE;
                    else if (c==FLAG) state= FLAG_STATE;
                    else state= START_STATE;
                    break;
                case BCC_STATE:
                    if (c==FLAG) {
                        state=END_STATE;
                        write(fd, buf_r, 5);
                    }
                    else state= START_STATE;
                    break;
                case END_STATE:
                    return 1;

            }
        }



        return 1;
    }

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


    printf("error");
    return -1;

}



unsigned char state_write(){
    enum State state = START_STATE;
    unsigned char read2, save;

    while (state!=END_STATE){
        if(read(fd, &read2, 1)>0){
            switch (state) {
                case START_STATE:
                    if (read2==FLAG)state=FLAG_STATE;
                    break;
                case FLAG_STATE:
                    if (read2==0x03) state=A_STATE;
                    else if (read2==FLAG) state=FLAG_STATE;
                    else state=START_STATE;
                    break;
                case A_STATE:
                    if (read2==RR0){
                        flagFrame=0;
                        save=read2;
                        state=C_STATE;
                    }
                    else if(read2==RR1){
                        flagFrame=1;
                        save=read2;
                        state=C_STATE;
```

```
                    }
                    else if(read2==REJ0){
                        flagFrame=0;
                        save=read2;
                        state=C_STATE;
                    }
                    else if (read2==REJ1){
                        flagFrame=1;
                        save=read2;
                        state=C_STATE;
                    }
                    else if (read2==DISC) save=read2;
                    else if (read2==FLAG) state=FLAG_STATE;
                    else state=START_STATE;
                    break;
                case C_STATE:
                    if (read2==(0x03^save))state= BCC_STATE;
                    else if (read2==FLAG) state=FLAG_STATE;
                    else state=START_STATE;
                    break;
                case BCC_STATE:
                    if (read2==FLAG)state = END_STATE;
                    else state=START_STATE;
                    break;
                default:
                    break;
            }

        }
        else break;
    }

    return save;

}




////////////////////////////////////////////////
// LLWRITE
////////////////////////////////////////////////
int llwrite(const unsigned char *buf, int bufSize)
{

    (void)signal(SIGALRM, alarmHandler);

    // TODO

    if (fd < 0) {
        perror("Invalid file descriptor");
        return -1;
    }

    unsigned char* stuffed = (unsigned char*)malloc(2 * bufSize * sizeof(unsigned char));
    stuffed[0]=FLAG;
    stuffed[1]=0x03;
    if (flagFrame==0){
        stuffed[2]=IN0;
    }
    else if (flagFrame==1){
        stuffed[2]=IN1;
    }
    stuffed[3]=stuffed[1]^stuffed[2];
```

```
316.     int cont=4;
317.     unsigned char xor=0;
318.     for(int i=0; i<bufSize; i++){
319.         xor^=buf[i];
320.         if (buf[i]==0x7e){
321.             stuffed[cont]=ESC;
322.             stuffed[cont+1]=0x5e;
323.             cont+=2;
324.         }
325.         else if (buf[i]==0x7d){
326.             stuffed[cont]=ESC;
327.             stuffed[cont+1]=0x5d;
328.             cont+=2;
329.         }
330.         else{
331.             stuffed[cont]=buf[i];
332.             cont++;
333.         }
334.     }
335.
336.
337.     stuffed[cont]=xor;
338.     stuffed[cont+1]=FLAG;
339.
340.     cont+=2;
341.
342.
343.     int trans = nRetransmissions;
344.
345.
346.     int bytes=0;
347.     alarmEnabled=FALSE;
348.     alarmCount=0;
349.
350.
351.     while (trans!=0){
352.         if (alarmEnabled == FALSE)
353.         {
354.             bytes = write(fd, stuffed, cont);
355.             alarm(timeout);
356.             alarmEnabled = TRUE;
357.         }
358.         if (bytes != cont) {
359.             fd=-1;
360.             perror("Error writing to serial port");
361.             return -1;
362.         }
363.         unsigned char read = state_write();
364.
365.         if (read==RR1 || read==RR0){
366.             alarm(0);
367.             break;
368.         }
369.         else if (read==REJ0 || read==REJ1){
370.             alarm(0);
371.         }
372.         alarmEnabled=FALSE;
373.         trans--;
374.
375.     }
376.
377.
378.     if (trans==0){
379.         return -1;
380.     }
```

```
381.
382.     return bytes;
383.
384.
385. }
386.
387. ////////////////////////////////////////////////
388. // LLREAD
389. ////////////////////////////////////////////////
390. int llread(unsigned char* packet)
391. {
392.
393.     unsigned char bytes;
394.     enum State state = START_STATE;
395.     unsigned char c_bye;
396.     int i=0;
397.
398.
399.     while (state!=END_STATE){
400.         if (read(fd, &bytes, 1)<0){
401.             return 1;
402.         }
403.         switch (state) {
404.             case START_STATE:
405.                 if (bytes == FLAG) state=FLAG_STATE;
406.                 break;
407.             case FLAG_STATE:
408.                 if (bytes==0x03) state=A_STATE;
409.                 else if (bytes==FLAG) state=FLAG_STATE;
410.                 else state=START_STATE;
411.                 break;
412.             case A_STATE:
413.                 if (bytes==IN0 || bytes==IN1){
414.                     state = C_STATE;
415.                     c_bye = bytes;
416.                     if (bytes==IN0) flagFrame=0;
417.                     else flagFrame=1;
418.                 }
419.                 else if (bytes==FLAG) state=FLAG_STATE;
420.                 else state=START_STATE;
421.                 break;
422.             case C_STATE:
423.                 if (bytes== (0x03^c_bye))state=DATA_STATE;
424.                 else if (bytes==FLAG) state=FLAG_STATE;
425.                 else state=START_STATE;
426.                 break;
427.             case DATA_STATE:
428.                 if (bytes==FLAG){
429.                     unsigned char BCC2 = packet[i-1];
430.                     unsigned char xor=0;
431.                     for (int j=0; j<i-1; j++){
432.                         xor^=packet[j];
433.                     }
434.
435.                     if (xor==BCC2) {
436.                         state = END_STATE;
437.                         if (flagFrame == 0) {
438.                             unsigned char send_RR1[5] = {FLAG, 0x03, RR1, 0x03 ^ RR1,
FLAG};
439.                             write(fd, send_RR1, 5);
440.                             return i-1;
441.                         }
442.                         else if (flagFrame==1){
443.                             unsigned char send_RR0[5] = {FLAG, 0x03, RR0, 0x03 ^ RR0,
FLAG};
```

```
444.                             write(fd, send_RR0, 5);
445.                             return i-1;
446.                         }
447.                     }
448.                     else{
449.                         printf("REJECTED\n");
450.                         if (flagFrame == 0) {
451.                             unsigned char send_REJ0[5] = {FLAG, 0x03, REJ0, 0x03 ^ REJ0,
FLAG};
452.                             write(fd, send_REJ0, 5);
453.                             return -1;
454.                         } else if (flagFrame == 1) {
455.                             unsigned char send_REJ1[5] = {FLAG, 0x03, REJ1, 0x03 ^ REJ1,
FLAG};
456.                             write(fd, send_REJ1, 5);
457.                             return -1;
458.                         }
459.                     }
460.                 }
461.                 else if (bytes==ESC) state = ESC_STATE;
462.                 else packet[i++]=bytes;
463.                 break;
464.             case ESC_STATE:
465.                 state=DATA_STATE;
466.                 if (bytes==0x5e) packet[i++]=0x7e;
467.                 else if (bytes==0x5d) packet[i++]=0x7d;
468.                 else{
469.                     packet[i++]=ESC;
470.                     packet[i++]=bytes;
471.                 }
472.                 break;
473.
474.         }
475.
476.     }
477.
478.     return -1;
479.
480. }
481.
482. ////////////////////////////////////////////////
483. // LLCLOSE
484. ////////////////////////////////////////////////
485. int llclose(int showStatistics)
486. {
487.
488.     (void)signal(SIGALRM, alarmHandler);
489.
490.     alarmEnabled=FALSE;
491.     alarmCount=0;
492.     unsigned char buf_disc[5] = {FLAG, 0x03, 0x0B, 0x03^0x0B, FLAG};
493.     unsigned char buf_ua[5] = {FLAG, 0x03, 0x07, 0x03^0x07, FLAG};
494.     unsigned char rec;
495.     int cont=0;
496.     unsigned char c;
497.
498.
499.     if (_role==LlTx) {
500.         (void) signal(SIGALRM, alarmHandler);
501.         enum State state = START_STATE;
502.
503.
504.         while (cont != nRetransmissions && state != END_STATE) {
505.
506.             if (alarmEnabled == FALSE) {
```

```
507.                int bytes = write(fd, buf_disc, 5);
508.                alarm(timeout);
509.                alarmEnabled = TRUE;
510.            }
511.            while (state != END_STATE) {
512.                read(fd, &c, 1);
513.                switch (state) {
514.                    case START_STATE:
515.                        if (c == FLAG) state = FLAG_STATE;
516.                        break;
517.                    case FLAG_STATE:
518.                        if (c == 0x01) state = A_STATE;
519.                        else if (c == FLAG) state = FLAG_STATE;
520.                        else state = START_STATE;
521.                        break;
522.                    case A_STATE:
523.                        if (c == DISC) state = C_STATE;
524.                        else if (c == FLAG) state = FLAG_STATE;
525.                        else state = START_STATE;
526.                        break;
527.                    case C_STATE:
528.                        if (c == (0x01 ^ 0x0B)) state = BCC_STATE;
529.                        else if (c == FLAG) state = FLAG_STATE;
530.                        else state = START_STATE;
531.                        break;
532.                    case BCC_STATE:
533.                        if (c == FLAG) {
534.                            state = END_STATE;
535.                            alarm(0);
536.                        } else state = START_STATE;
537.                        break;
538.
539.
540.                }
541.            }
542.            alarmEnabled = FALSE;
543.            cont++;
544.        }
545.
546.        write(fd, buf_ua, 5);
547.    }
548.
549.    else if (_role==LlRx) {
550.        unsigned char bytes;
551.        enum State state_r = START_STATE;
552.        unsigned char c_bye;
553.        int i = 0;
554.
555.        unsigned char sv;
556.
557.        while (state_r != END_STATE) {
558.
559.            read(fd, &bytes, 1);
560.            printf("%X\n", bytes);
561.
562.            switch (state_r) {
563.                case START_STATE:
564.                    if (bytes == FLAG) state_r = FLAG_STATE;
565.                    break;
566.                case FLAG_STATE:
567.                    if (bytes == 0x03) state_r = A_STATE;
568.                    else if (bytes == FLAG) state_r = FLAG_STATE;
569.                    else state_r = START_STATE;
570.                    break;
571.                case A_STATE:
```

```
572.                     if (bytes == DISC) {
573.                         sv=bytes;
574.                         state_r=C_STATE;
575.                     } else if (bytes == FLAG) state_r = FLAG_STATE;
576.                     else if (bytes==0x07){
577.                         state_r=C_STATE;
578.                         sv=bytes;
579.                     }
580.                     else state_r = START_STATE;
581.                     break;
582.                 case C_STATE:
583.                     if (bytes== (0x03^sv))state_r=BCC_STATE;
584.                     else state_r=START_STATE;
585.                     break;
586.                 case BCC_STATE:
587.                     if (bytes==FLAG){
588.                         if (sv==DISC){
589.                             unsigned char send_disc[5] = {FLAG, 0x01, DISC, 0x01 ^ DISC,
FLAG};
590.                             printf("DISC RECEIVED\n");
591.                             write(fd, send_disc, 5);
592.                             state_r = START_STATE;
593.                         }
594.                         else if (sv==0x07){
595.                             printf("ENTERING DISC\n");
596.                             state_r=END_STATE;
597.                         }
598.                     }
599.                     break;
600.
601.             }
602.         }
603.
604.     }
605.
606.     if (tcsetattr(fd, TCSANOW, &oldtio) == -1)
607.     {
608.         perror("tcsetattr");
609.         exit(-1);
610.     }
611.     return close(fd);
612.
613. }
614.
```

main.c

```c
// Main file of the serial port project.
// NOTE: This file must not be changed.

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

#include "application_layer.h"

#define BAUDRATE 9600
#define N_TRIES 3
#define TIMEOUT 4

// Arguments:
//   $1: /dev/ttySxx
//   $2: tx | rx
//   $3: filename
int main(int argc, char *argv[])
{
    if (argc < 4)
    {
        printf("Usage: %s /dev/ttySxx tx|rx filename\n", argv[0]);
        exit(1);
    }

    const char *serialPort = argv[1];
    const char *role = argv[2];
    const char *filename = argv[3];

    printf("Starting link-layer protocol application\n"
           "  - Serial port: %s\n"
           "  - Role: %s\n"
           "  - Baudrate: %d\n"
           "  - Number of tries: %d\n"
           "  - Timeout: %d\n"
           "  - Filename: %s\n",
           serialPort,
           role,
           BAUDRATE,
           N_TRIES,
           TIMEOUT,
           filename);

    applicationLayer(serialPort, role, BAUDRATE, N_TRIES, TIMEOUT, filename);

    return 0;
}

```