Protocolo de Ligação de Dados

RCOM TP1

by Gustavo Costa, LEIC02

Índice

Sumário **4**

Introdução **4**

Arquitetura **5**

Link Layer 5

Application Layer 5

Estrutura do código **6**

link_layer.c/h 6

application_layer.c/h 6

auxil.c/h 6

macros.h 6

state_machines.h 6

Use cases principais **7**

Transmissor 7

Recetor 7

Protocolo Link Layer **8**

int llopen(LinkLayer connectionParameters); 8

int llwrite(const unsigned char *buf, int bufSize); 8

int llread(unsigned char *packet); 8

int llclose(int showStatistics); 8

Mecanismo de retransmissões Stop and Wait 9

Máquinas de estados 9

Protocolo Application Layer **9**

void applicationLayer(const char *serialPort, const char *role, int baudRate, int nTries, int timeout, const char *filename); 9

Validação **10**

Eficiência do protocolo link layer **11**

Variação do tamanho das tramas enviadas 11

Variação da capacidade da ligação 11

Variação do tempo de propagação 12

Geração de erros nas tramas de informação 12

Como é possível perceber, uma pequena percentagem de erros diminuiu drasticamente a eficiência do protocolo. Estes resultados não são necessariamente corretos, pois o número de testes efetuados foi relativamente baixo, podendo ter havido “azar” a correr os primeiros dois testes, justificando assim uma descida tão drástica. 13

Conclusões **14**

Anexo I - Código fonte **15**

link_layer.c 15

auxil.c 33
application_layer.c 36
state_machines.h 40
macros.h 40

Sumário

O TP1 remete à transferência de ficheiros através da porta de série RS-232, tendo em conta todos os passos necessários, desde a abertura da conexão até ao tratamento e processamento dos dados enviados/recebidos, introduzindo protocolos de correção redundância.

Sendo possível transferir ficheiros sem perdas, mesmo impedindo temporariamente a transmissão, dou o objetivo do trabalho como alcançado.

Introdução

O relatório terá a seguinte estrutura, com o intuito de esclarecer o objetivo de cada um dos tópicos, assim como revelar a forma e funcionamento do trabalho que foi feito até agora:

- **Arquitetura** - blocos funcionais e interfaces.
- **Estrutura do código** - APIs, principais estruturas de dados, principais funções e a sua relação com a arquitetura.
- ***Use cases* principais** - identificação; sequências de chamada de funções.
- **Protocolo *Link Layer*** - identificação dos principais aspetos funcionais; descrição da estratégia de implementação destes aspetos com apresentação de extratos de código.
- **Protocolo *Application Layer*** - identificação dos principais aspetos funcionais; descrição da estratégia de implementação destes aspetos com apresentação de extratos de código.
- **Validação** - descrição dos testes efetuados.
- **Eficiência do protocolo *link layer*** - caracterização estatística da eficiência do protocolo, efetuada recorrendo a medidas sobre o código desenvolvido.
- **Conclusões** - síntese da informação apresentada nas secções anteriores; reflexão sobre os objetivos de aprendizagem alcançados.

Arquitetura

De acordo com o pedido, o projeto implementa a camada de ligação (*Link Layer* ou *Data Link Layer*) e uma camada de aplicação de teste (*Application Layer*), sendo cada uma delas responsável pela implementação e gestão do protocolo associado.

Ambas são bem definidas e distintas, permitindo a sua troca por outras que suportem a mesma API.

Link Layer

Camada responsável por iniciar e terminar a ligação que permite a transmissão e recepção de dados entre dois computadores via porta de série, assim como efetuar as mesmas.

Application Layer

Camada responsável por preparar o ficheiro a ser transmitido, receber o ficheiro transmitido e utilizar as funcionalidades da camada anterior para efetuar a transferência do mesmo, de acordo com parâmetros que podem ser alterados do ponto de vista do utilizador.

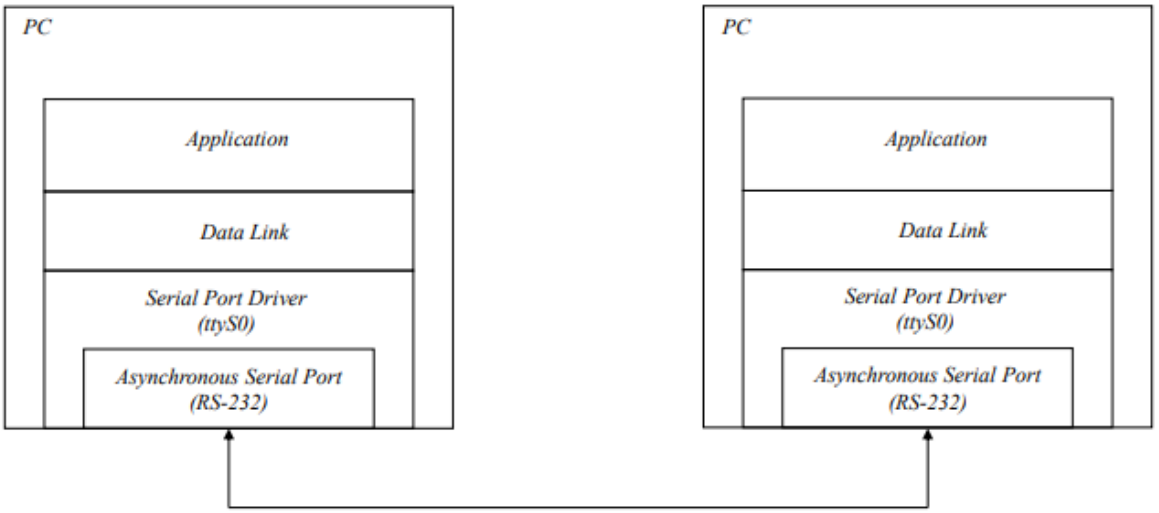

Diagrama representativo da arquitetura do projeto.

Estrutura do código

O código é composto por dois ficheiros principais: application_layer.c/h e link_layer.c/h, sendo cada um deles responsável pelo respetivo protocolo.

Existem também alguns ficheiros auxiliares que visam facilitar a implementação de algumas funcionalidades: auxil.c/h, macros.h, state_machines.h.

As funções contidas nos ficheiros e as suas descrições são as seguintes:

link_layer.c/h

- llopen - inicia a ligação entre o transmissor e o recetor
- llwrite - recebe os dados da *Application layer,* monta a trama e envia para o recetor
- llread - recebe a trama do transmissor, desmonta a trama e envia os dados para a *Application layer*
- llclose - fecha a ligação entre o transmissor e o recetor

application_layer.c/h

- applicationLayer - além de chamar todas as funções anteriormente listadas, nessa mesma ordem, divide o ficheiro a ser transmitido em blocos, e, no caso do recetor, monta o ficheiro recebido.

auxil.c/h

- buildControlPacket - função que monta os *packets* de controlo da informação
- buildDataPacket - função que monta os *packets* de dados da informação
- byteStuff - função que dá *byte stuffing* dos dados
- byteDestuff - função que dá *byte de-stuffing* dos dados
- calculateBCC2 - função que calcula o BCC2 dos dados recebidos

macros.h

Não contém funções, apenas macros úteis à realização do trabalho.

state_machines.h

Não contém funções, apenas uma *struct* útil à realização do trabalho.

Use cases principais

A sequência de chamada de funções varia, dependendo do papel representado por cada computador. Assim sendo, descreve-se esta sequência do ponto de vista do transmissor e do recetor.

Transmissor

1. Faz um pedido de ligação ao recetor. (llopen)
2. Abre o ficheiro a enviar.
3. Envia a trama de informação com o *packet* de controlo a indicar o início da transmissão de dados. (llwrite)
4. Lê e envia blocos de dados do ficheiro até este ter sido totalmente enviado. (llwrite)
5. Envia a trama de informação com o *packet* de controlo a indicar o fim da transmissão de dados. (llwrite)
6. Anuncia o término da ligação. (llclose)

Recetor

1. Acusa a receção do pedido de ligação. (llopen)
2. Cria um ficheiro vazio para guardar os dados.
3. Recebe a trama de informação com o *packet* de controlo a indicar o início da transmissão de dados. (llread)
4. Recebe e guarda blocos de dados do ficheiro até este ter sido totalmente recebido. (llread)
5. Recebe a trama de informação com o *packet* de controlo a indicar o fim da transmissão de dados. (llread)
6. Acusa a receção do término de ligação. (llclose)

Protocolo *Link Layer*

O protocolo *Link Layer* foi implementado de acordo com a especificação pedida.

int llopen(LinkLayer connectionParameters);

A função llopen tem como objetivo iniciar a ligação entre o transmissor e o recetor.

Nos argumentos desta função são passados os parâmetros da ligação, como o nr de retransmissões, o tempo do *timeout*, a *role* da porta de série, etc

Caso quem chame esta função seja o transmissor, a função enviará uma trama de controlo do tipo SET e aguardará por uma resposta válida do tipo UA (que também é uma trama de controlo), ativando o alarme, caso seja necessário pôr em prática o mecanismo de retransmissões. Mal seja recebida a resposta correta, a ligação é considerada como estabelecida.

Em contrapartida, se quem chamar esta função for o recetor, a função aguardará até receber uma trama de controlo do tipo SET e, mal o faça envia uma trama de controlo do tipo UA.

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

A função llwrite tem como objetivo o envio de dados do transmissor para o recetor.

Nos argumentos desta função é passado os dados a serem enviados e o tamanho dos mesmos.

Esta função é responsável por dar *byte stuff* à informação a ser enviada (função byteStuff), montar a trama de informação e aguardar por uma resposta positiva por parte do recetor.

Caso esta resposta nunca chegue, reenvia os dados, utilizando o mecanismo de retransmissões.

int llread(unsigned char *packet);

A função llread tem como objetivo receber os dados enviados pelo transmissor.

Nos argumentos desta função é passado o *packet* onde deverão ser guardados os dados recebidos, caso estejam corretos.

Esta função é responsável por dar *byte de-stuff* da informação (função byteDestuff) , verificar se esta se encontra correta (função calculateBCC2 e verificação do nr de sequência) e enviar uma resposta ao transmissor, de acordo com a validade da informação.

int llclose(int showStatistics);

A função llclose tem como objetivo fechar a ligação entre o transmissor e o recetor.

Nos argumentos desta função é passado um *booleano* cujo valor representa a vontade que o utilizador tem em saber as estatísticas da transmissão.

Caso quem chame esta função seja o transmissor, a função enviará uma trama de controlo do tipo DISC e aguardará por uma trama do tipo DISC à qual responderá com uma trama de resposta do tipo UA, fechando a conexão.

Em contrapartida, se quem chamar esta função for o recetor, a função aguardará até receber uma trama de controlo do tipo DISC e, mal o faça envia uma trama de controlo do tipo DISC, passando a aguardar por uma do tipo UA, que poderá nunca chegar.

Mecanismo de retransmissões *Stop and Wait*

O mecanismo de retransmissões *Stop and Wait* está implementado e suportado por todas as funções da *Link Layer*, exceto a função llread, que não necessita.

Este mecanismo permite o reenvio de informação, até um número máximo de vezes pré-definido, mediante falhas na conexão, tais como distúrbios na conexão que gerem erros na informação ou interrupções de conexão que impossibilitem a transferência de dados até um tempo máximo pré-definido.

Máquinas de estados

O processamento de tramas é efetuado utilizando máquinas de estados, o que permite a validação da informação recebida sequencialmente, garantindo que o que se estar a ler é, de facto, o que se pensa estar a ler.

Protocolo *Application Layer*

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

A função applicationLayer tem como objetivo testar a API da *Link Layer* chamando todas as funções desta, assim como dividir o ficheiro que será transmitido em blocos e montar o ficheiro recebido.

Nos argumentos desta função são passados os parâmetros da ligação, assim como o nome do ficheiro a ser transmitido.

Após a chamada da função llopen, dependendo da *role* da porta de série, uma de duas coisas pode acontecer:

1. Transmissor:
    a. O *packet* que sinaliza o início do envio de dados é montado (função buildControlPacket) e enviado através da primeira chamada à função llwrite.
    b. Consequentemente, o ficheiro é dividido em blocos que são montados em *data packets* (função buildDataPacket) etransmitidos com recurso a chamadas sequenciais à função llwrite.
    c. Aquando do término da transmissão do ficheiro, o *packet* que sinaliza o fim do envio de dados é montado (função buildControlPacket) e enviado através da última chamada à função llwrite.
2. Recetor:
    a. É aberto o ficheiro onde será guardada a informação recebida.
    b. São feitas chamadas sequenciais à função llread, lendo o *packet* devolvido e escrevendo-o no ficheiro, caso sejam dados.

Por fim, e se tudo correu de acordo com o esperado, é chamada a função llclose para reverter a porta de série para as definições base, encerrando a conexão.

Validação

No momento da avaliação presencial, foram efetuados, pelo professor, os seguintes testes:

1. Envio normal do ficheiro.
2. Início da transmissão com a porta de série desligada, ligando-a a meio da transmissão.
3. Início da transmissão com a porta de série ligada, desligando-a e voltando a ligá-la a meio da transmissão.
4. Introdução de erros, gerando curto circuito na porta de série.
5. Alteração do tamanho dos pacotes a serem transmitidos para um valor superior e inferior ao *default*.

Eficiência do protocolo *link layer*

A modo de verificar a eficiência do protocolo, foram efetuados 4 testes aos parâmetros: tamanho das tramas, capacidade da ligação, tempo de propagação e erros gerados na transmissão.

Variação do tamanho das tramas enviadas

Fixando o *baudrate (C)* a 38400 e enviando um ficheiro de 10968 bytes, foi variado o tamanho das tramas enviadas.

Na seguinte tabela são apresentados os resultados obtidos para os vários tamanhos:

Tamanho (Bytes)	Tempo (s)	Rate (bits/s)	S (R/C)
64	3.572	24564.3897	0.64
128	3.226	27199.0080	0.71
256	3.053	28740.2555	0.75
512	2.969	29553.3850	0.77
1024	2.925	29997.9487	0.78
2048	2.905	30204.4750	0.79

Como é possível perceber, o aumento da eficiência do protocolo é proporcional ao aumento do tamanho das tramas enviadas.

Variação da capacidade da ligação

Fixando o tamanho de uma trama a 1000 bytes e enviando um ficheiro de 10968 bytes, foi variado o *baudrate*.

Na seguinte tabela são apresentados os resultados obtidos para os vários *baudrates*:

Baudrate	Tempo (s)	Rate (bits/s)	S (R/C)
4800	23.389	3751.5071	0.78
9600	11.695	7502.6935	0.78
19200	5.848	15004.1040	0.78
38400	2.925	29997.9487	0.78
57600	1.950	44996.9231	0.78
115200	0.976	89901.6393	0.78

Como é possível perceber, a eficiência do protocolo mantém-se constante, independentemente do *baudrate*.

Variação do tempo de propagação

Este teste foi feito acrescentando tempo de propagação através da função sleep() e fixando o baudrate em 38400, o tamanho da trama em 1000bytes e enviando um ficheiro de 10968bytes.

Na seguinte tabela são apresentados os resultados obtidos para os vários tempos adicionados:

Tempo prop (s)	Tempo (s)	Rate (bits/s)	S (R/C)
0	2.925	29997.9487	0.78
0.01	3.187	27531.8481	0.72
0.1	5.527	15874.5202	0.41
0.5	15.927	5509.1354	0.14
1	28.927	3033.2907	0.08
1.5	41.927	2092.7803	0.05

Como é possível perceber, à medida que o tempo de propagação aumenta, a eficiência diminui drasticamente.

Geração de erros nas tramas de informação

Sendo introduzido um erro no BCC2 a cada N tramas de informação, calculei a eficiência em função do número de erros, fixando o *baudrate* em 38400, o tamanho da trama em 1000 bytes e enviando um ficheiro de 10968 bytes.

Na seguinte tabela são apresentados os resultados obtidos para os vários erros, tendo corrido cada teste 10 vezes:

N	% de erros (100/N)	Tempo (s)	Rate (bits/s)	S (R/C)
10	10	6.927	12666.9554	0.33
20	5	3.924	22360.8563	0.58
50	2	3.042	28844.1815	0.75
100	1	2.925	29997.9487	0.78
1000	0.1	2.924	30008.2079	0.78
10000	0.01	2.925	29997.9487	0.78

Como é possível perceber, uma pequena percentagem de erros diminuiu drasticamente a eficiência do protocolo. Estes resultados não são necessariamente corretos, pois o número de testes efetuados foi relativamente baixo, podendo ter havido "azar" a correr os primeiros dois testes, justificando assim uma descida tão drástica.

Conclusões

A realização deste trabalho, que visava a implementação do protocolo de ligação de dados, permitiu o melhor conhecimento dos conceitos e estruturas relativas a este tópico.

Conceitos como a distinção entre as camadas que participam numa transmissão de dados, o mecanismo de retransmissão Stop and Wait, os mecanismos de verificação de dados ficaram melhor interiorizados, devido à realização do trabalho.

Concluo assim que, apesar de ter lidado com algumas adversidades, o projeto foi uma mais valia, devido à ultrapassagem das mesmas e ao conhecimento adquirido.

Anexo I - Código fonte

link_layer.c

```
// Link layer protocol implementation

#include "link_layer.h"
#include "state_machines.h"
#include "macros.h"
#include "auxil.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>


#define FALSE 0
#define TRUE 1

int alarmEnabled = FALSE;
int alarmCount = 0; Ns = 0; Nr = 1;


// Baudrate settings are defined in <asm/termbits.h>, which is
// included by <termios.h>
#define BAUDRATE B38400
#define _POSIX_SOURCE 1 // POSIX compliant source

#define BUF_SIZE 256

int fd;
struct termios oldtio;
struct termios newtio;
LinkLayer connectionParameters2;

// Alarm function handler
void alarmHandler(int signal)
{
    alarmEnabled = FALSE;
    alarmCount++;
```

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

//Resets alarm counter and disables alarm
void resetAlarm(){
    alarmEnabled = FALSE;
    alarmCount = 0;
}

////////////////////////////////////////////////
// LLOPEN
////////////////////////////////////////////////
int llopen(LinkLayer connectionParameters)
{
    connectionParameters2 = connectionParameters;
    const char *serialPortName = connectionParameters.serialPort;

    // Open serial port device for reading and writing and not as
controlling tty
    // because we don't want to get killed if linenoise sends CTRL-C.
    fd = open(serialPortName, O_RDWR | O_NOCTTY);
    if (fd < 0)
    {
        perror(serialPortName);
        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 = 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] = 0.1; // 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);
    }

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

    unsigned char buf[BUF_SIZE + 1] = {0}; // +1: Save space for the
final '\0' char

    switch (connectionParameters.role){
    //Receiver
    case LlRx:{
        /* Read até SET estar feito */
        state_machine stateMachineState = Start;
        while (stateMachineState != STOP)
        {
            //read 1 byte at a time
            if(read(fd, buf, 1)==0){
                continue;
            }
            // Set state machine
            switch (stateMachineState)
            {
            case Start:
                if(buf[0]==FLAG){
                    stateMachineState = FLAG_RCV;
                }
                else{
                    stateMachineState = Start;
                }
                break;
```

```
            case FLAG_RCV:
                if(buf[0]==FLAG){
                    stateMachineState = FLAG_RCV;
                }
                else if(buf[0]==AddrTR){
                    stateMachineState = A_RCV;
                }
                else{
                    stateMachineState = Start;
                }
                break;
            case A_RCV:
                if(buf[0]==FLAG){
                    stateMachineState = FLAG_RCV;
                }
                else if(buf[0]==SET){
                    stateMachineState = C_RCV;
                }
                else{
                    stateMachineState = Start;
                }
                break;
            case C_RCV:
                if(buf[0]==FLAG){
                    stateMachineState = FLAG_RCV;
                }
                else if(buf[0]==(AddrTR ^ SET)){
                    stateMachineState = BCC_OK;
                }
                else{
                    stateMachineState = Start;
                }
                break;
            case BCC_OK:
                if(buf[0]==FLAG){
                    stateMachineState = STOP;
                }
                else{
                    stateMachineState = Start;
                }
                break;
            default:
                break;
            }
        }
```

```
        /* Montar UA */
        char ua[BUF_SIZE] = {0};
        ua[0] = FLAG;
        ua[1] = AddrTR;
        ua[2] = UA;
        ua[3] = AddrTR ^ UA;
        ua[4] = FLAG;

        /* Enviar UA */
        write(fd, ua, 5);
        break;
    }
    //Transmiter
    case LlTx:{

        /* Setup alarm */
        (void)signal(SIGALRM, alarmHandler);

        /* Montar Set */
        char set[BUF_SIZE] = {0};
        set[0] = FLAG;
        set[1] = AddrTR;
        set[2] = SET;
        set[3] = AddrTR ^ SET;
        set[4] = FLAG;

        int nr_retrans = 0;

        while(nr_retrans < connectionParameters.nRetransmissions){

            alarm(connectionParameters.timeout); // Set alarm to be
triggered in defined time

            /* Enviar Set */
            write(fd, set, 5);

            alarmEnabled = TRUE;

            /* Read até UA estar feito e repetir até ter UA ou nr max de
retransmissões excedido */
            state_machine stateMachineState = Start;

            while (stateMachineState != STOP && alarmEnabled){
                //read 1 byte at a time
                if(read(fd, buf, 1)==0){
                    continue;
```

```
            }

            switch (stateMachineState)
            {
            case Start:
                if(buf[0]==FLAG){
                    stateMachineState = FLAG_RCV;
                }
                else{
                    stateMachineState = Start;
                }
                break;
            case FLAG_RCV:
                if(buf[0]==FLAG){
                    stateMachineState = FLAG_RCV;
                }
                else if(buf[0]==AddrTR){
                    stateMachineState = A_RCV;
                }
                else{
                    stateMachineState = Start;
                }
                break;
            case A_RCV:
                if(buf[0]==FLAG){
                    stateMachineState = FLAG_RCV;
                }
                else if(buf[0]==UA){
                    stateMachineState = C_RCV;
                }
                else{
                    stateMachineState = Start;
                }
                break;
            case C_RCV:
                if(buf[0]==FLAG){
                    stateMachineState = FLAG_RCV;
                }
                else if(buf[0]==(AddrTR ^ UA)){
                    stateMachineState = BCC_OK;
                }
                else{
                    stateMachineState = Start;
                }
                break;
            case BCC_OK:
```

```
                    if(buf[0]==FLAG){
                        stateMachineState = STOP;
                    }
                    else{
                        stateMachineState = Start;
                    }
                    break;
                default:
                    break;
                }
            }
            if(stateMachineState == STOP){
                resetAlarm();
                break;
            }
            nr_retrans++;
        }
        if(nr_retrans==connectionParameters.nRetransmissions){
            // Restore the old port settings
            if (tcsetattr(fd, TCSANOW, &oldtio) == -1)
            {
                perror("tcsetattr");
                exit(-1);
            }

            close(fd);
            exit(-1);
        }
        break;
    }
    default:
        break;
    }
    printf("Connection established\n");
    return 1;
}

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

    int NR = changeN(Ns);
```

```
    /* Montar IFrame */
    unsigned char info[bufSize*2 + 6]; // Assuming all bytes need
stuffing
    info[0] = FLAG; //Flag
    info[1] = AddrTR; //Address
    info[2] = CONTROL(Ns); //Control
    info[3] = info[1] ^ info[2]; //BCC1

    int nr_retrans = 0, size = 4; int bcc2 = 0;

    unsigned char buf2[5];

    byteStuff(&info, &size, &bcc2, bufSize, buf);

    while(nr_retrans < connectionParameters2.nRetransmissions){

        alarm(connectionParameters2.timeout);

        /* Enviar IFrame */
        write(fd, info, size);

        alarmEnabled = TRUE;

        state_machine stateMachineState = Start;
        while (stateMachineState != STOP && alarmEnabled){

            //read 1 byte at a time
            if(read(fd, buf2, 1)==0){
                continue;
            }

            switch (stateMachineState)
            {
            case Start:
                if(buf2[0]==FLAG){
                    stateMachineState = FLAG_RCV;
                }
                else{
                    stateMachineState = Start;
                }
                break;
            case FLAG_RCV:
                if(buf2[0]==FLAG){
                    stateMachineState = FLAG_RCV;
                }
                else if(buf2[0]==AddrTR){
```

```
                stateMachineState = A_RCV;
            }
            else{
                stateMachineState = Start;
            }
            break;
        case A_RCV:
            if(buf2[0]==FLAG){
                stateMachineState = FLAG_RCV;
            }
            else if(buf2[0]==PosACK(NR)){
                stateMachineState = C_RCV;
            }
            else if(buf2[0]==NegACK(NR)){
                stateMachineState = STOP;
                if(read(fd, buf2, 1)==0){
                    continue;
                }
                if(read(fd, buf2, 1)==0){
                    continue;
                }
            }
            else{
                stateMachineState = Start;
            }
            break;
        case C_RCV:
            if(buf2[0]==FLAG){
                stateMachineState = FLAG_RCV;
            }
            else if(buf2[0]==(AddrTR ^ PosACK(NR))){
                stateMachineState = BCC_OK;
            }
            else{
                stateMachineState = Start;
            }
            break;
        case BCC_OK:
            if(buf2[0]==FLAG){
                stateMachineState = STOP;
            }
            else{
                stateMachineState = Start;
            }
            break;
        default:
```

```
                break;
            }
        }
        if(stateMachineState == STOP){
            resetAlarm();
            break;
        }
        nr_retrans++;
    }
    if(nr_retrans==connectionParameters2.nRetransmissions){
        // Restore the old port settings
        if (tcsetattr(fd, TCSANOW, &oldtio) == -1)
        {
            perror("tcsetattr");
            exit(-1);
        }

        close(fd);
        exit(-1);
    }

    Ns= (Ns+1)%2;
    return size;

}

////////////////////////////////////////////////
// LLREAD
////////////////////////////////////////////////
int llread(unsigned char *packet)
{

    unsigned char buf[2*MAX_PAYLOAD_SIZE+6];
    int size = 0;

    int NS = changeN(Nr);

    state_machine stateMachineState = Start;
    while (stateMachineState != STOP)
    {
        //read 1 byte at a time
        if(read(fd, buf+size, 1) <= 0){
            continue;
        }

        switch (stateMachineState)
```

```
        {
        case Start:
            if(buf[size]==FLAG){
                stateMachineState = FLAG_RCV;
                size++;
            }
            else{
                stateMachineState = Start;
            }
            break;
        case FLAG_RCV:
            if(buf[size]==FLAG){
                stateMachineState = FLAG_RCV;
            }
            else if(buf[size]==AddrTR){
                stateMachineState = A_RCV;
                size++;
            }
            else{
                stateMachineState = Start;
            }
            break;
        case A_RCV:
            if(buf[size]==FLAG){
                stateMachineState = FLAG_RCV;
            }
            else if(buf[size]==CONTROL(NS)){
                stateMachineState = C_RCV;
                size++;
            }
            else{
                stateMachineState = Start;
            }
            break;
        case C_RCV:
            if(buf[size]==FLAG){
                stateMachineState = FLAG_RCV;
            }
            else if(buf[size]==(AddrTR ^ CONTROL(NS))){
                stateMachineState = BCC_OK;
                size++;
            }
            else{
                stateMachineState = Start;
            }
            break;
```

```
        case BCC_OK:
            if(buf[size]==FLAG){
                stateMachineState = STOP;
            }
            size++;
            break;
        default:
            break;
        }
    }

    byteDestuff(packet, &size, &buf);

    int bcc2 = calculateBCC2(packet, size);

    if(bcc2 != packet[size-1]){

        //Montar REJ
        unsigned char REJ[5];
        REJ[0] = FLAG; //Flag
        REJ[1] = AddrTR; //Address
        REJ[2] = NegACK(Nr); //Control
        REJ[3] = AddrTR ^ NegACK(Nr); //BCC1
        REJ[4] = FLAG;

        fprintf(stdout, "Sending REJ\n");
        write(fd, REJ, 5);

        return -1;
    }

    packet[size-1] = '\0';

    //Montar RR
    unsigned char RR[5];
    RR[0] = FLAG; //Flag
    RR[1] = AddrTR; //Address
    RR[2] = PosACK(Nr); //Control
    RR[3] = AddrTR ^ PosACK(Nr); //BCC1
    RR[4] = FLAG;

    fprintf(stdout, "Sending RR\n");
    write(fd, RR, 5);

    Nr=(Nr+1)%2;
```

```
    return size;
}

////////////////////////////////////////////////
// LLCLOSE
////////////////////////////////////////////////
int llclose(int showStatistics)
{

    printf("Severing connection\n");
    unsigned char buf[BUF_SIZE + 1] = {0};

    switch (connectionParameters2.role){
    //Receiver
    case LlRx:{
        /* Read até DISC estar feito */
        state_machine stateMachineState = Start;
        while (stateMachineState != STOP)
        {
            //read 1 byte at a time
            if(read(fd, buf, 1)==0){
                continue;
            }

            switch (stateMachineState)
            {
            case Start:
                if(buf[0]==FLAG){
                    stateMachineState = FLAG_RCV;
                }
                else{
                    stateMachineState = Start;
                }
                break;
            case FLAG_RCV:
                if(buf[0]==FLAG){
                    stateMachineState = FLAG_RCV;
                }
                else if(buf[0]==AddrTR){
                    stateMachineState = A_RCV;
                }
                else{
                    stateMachineState = Start;
                }
                break;
            case A_RCV:
```

```
            if(buf[0]==FLAG){
                stateMachineState = FLAG_RCV;
            }
            else if(buf[0]==DISC){
                stateMachineState = C_RCV;
            }
            else{
                stateMachineState = Start;
            }
            break;
        case C_RCV:
            if(buf[0]==FLAG){
                stateMachineState = FLAG_RCV;
            }
            else if(buf[0]==(AddrTR ^ DISC)){
                stateMachineState = BCC_OK;
            }
            else{
                stateMachineState = Start;
            }
            break;
        case BCC_OK:
            if(buf[0]==FLAG){
                stateMachineState = STOP;
            }
            else{
                stateMachineState = Start;
            }
            break;
        default:
            break;
        }
    }

    /* Montar DISC */
    char ua[BUF_SIZE] = {0};
    ua[0] = FLAG;
    ua[1] = AddrRT;
    ua[2] = DISC;
    ua[3] = AddrRT ^ DISC;
    ua[4] = FLAG;

    /* Enviar DISC */
    write(fd, ua, 5);

    /* Read até UA estar feito */
```

```
        stateMachineState = Start;
        while (stateMachineState != STOP)
        {
            //read 1 byte at a time
            if(read(fd, buf, 1)==0){
                continue;
            }

            switch (stateMachineState)
            {
            case Start:
                if(buf[0]==FLAG){
                    stateMachineState = FLAG_RCV;
                }
                else{
                    stateMachineState = Start;
                }
                break;
            case FLAG_RCV:
                if(buf[0]==FLAG){
                    stateMachineState = FLAG_RCV;
                }
                else if(buf[0]==AddrTR){
                    stateMachineState = A_RCV;
                }
                else{
                    stateMachineState = Start;
                }
                break;
            case A_RCV:
                if(buf[0]==FLAG){
                    stateMachineState = FLAG_RCV;
                }
                else if(buf[0]==UA){
                    stateMachineState = C_RCV;
                }
                else{
                    stateMachineState = Start;
                }
                break;
            case C_RCV:
                if(buf[0]==FLAG){
                    stateMachineState = FLAG_RCV;
                }
                else if(buf[0]==(AddrTR ^ UA)){
                    stateMachineState = BCC_OK;
```

```
                }
                else{
                    stateMachineState = Start;
                }
                break;
            case BCC_OK:
                if(buf[0]==FLAG){
                    stateMachineState = STOP;
                }
                else{
                    stateMachineState = Start;
                }
                break;
            default:
                break;
            }
        }

        break;
    }
    //Transmiter
    case LlTx:{

        resetAlarm();

        /* Setup alarm */
        (void)signal(SIGALRM, alarmHandler);

        /* Montar DISC */
        char set[BUF_SIZE] = {0};
        set[0] = FLAG;
        set[1] = AddrTR;
        set[2] = DISC;
        set[3] = AddrTR ^ DISC;
        set[4] = FLAG;

        int nr_retrans = 0;

        unsigned char buf[5]={0};

        while(nr_retrans < connectionParameters2.nRetransmissions){

            alarm(connectionParameters2.timeout); // Set alarm to be
triggered in defined time

            /* Enviar DISC */
```

```
            write(fd, set, 5);

            alarmEnabled = TRUE;

            /* Read até DISC estar feito e repetir até ter DISC ou nr
max de retransmissões excedido */
            state_machine stateMachineState = Start;

            while (stateMachineState != STOP && alarmEnabled)
            {
                //read 1 byte at a time
                if(read(fd, buf, 1)==0){
                    continue;
                }

                switch (stateMachineState)
                {
                case Start:
                    if(buf[0]==FLAG){
                        stateMachineState = FLAG_RCV;
                    }
                    else{
                        stateMachineState = Start;
                    }
                    break;
                case FLAG_RCV:
                    if(buf[0]==FLAG){
                        stateMachineState = FLAG_RCV;
                    }
                    else if(buf[0]==AddrRT){
                        stateMachineState = A_RCV;
                    }
                    else{
                        stateMachineState = Start;
                    }
                    break;
                case A_RCV:
                    if(buf[0]==FLAG){
                        stateMachineState = FLAG_RCV;
                    }
                    else if(buf[0]==DISC){
                        stateMachineState = C_RCV;
                    }
                    else{
                        stateMachineState = Start;
                    }
```

```
                break;
            case C_RCV:
                if(buf[0]==FLAG){
                    stateMachineState = FLAG_RCV;
                }
                else if(buf[0]==(AddrRT ^ DISC)){
                    stateMachineState = BCC_OK;
                }
                else{
                    stateMachineState = Start;
                }
                break;
            case BCC_OK:
                if(buf[0]==FLAG){
                    stateMachineState = STOP;
                }
                else{
                    stateMachineState = Start;
                }
                break;
            default:
                break;
            }
        }
        if(stateMachineState == STOP){
            resetAlarm();
            break;
        }
        nr_retrans++;
    }
    if(nr_retrans==connectionParameters2.nRetransmissions){
        // Restore the old port settings
        if (tcsetattr(fd, TCSANOW, &oldtio) == -1)
        {
            perror("tcsetattr");
            exit(-1);
        }

        close(fd);
        exit(-1);
    }

   /* Montar UA */
    char ua[BUF_SIZE] = {0};
    ua[0] = FLAG;
    ua[1] = AddrTR;
```

```
        ua[2] = UA;
        ua[3] = AddrTR ^ UA;
        ua[4] = FLAG;

        /* Enviar UA */
        write(fd, ua, 5);

        break;
    }
    default:
        break;
    }

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

    close(fd);

    printf("Connection severed\n");

    return 1;
}
```

auxil.c

```
#include "auxil.h"
#include "macros.h"

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

int changeN(int N){
    if(N == 1) return 0;
    else return 1;
}

int buildControlPacket(unsigned char *control_packet, int control,
unsigned int fileSize, char *filename){

    control_packet[0] = control; // Control field
```

```
    control_packet[1] = FILE_SIZE; // T1 Field for fileSize

    unsigned int aux = fileSize, byteCount = 0;
    while(aux != 0){
        aux >>= 8;
        byteCount++;
    }

    unsigned int L1 = byteCount;
    control_packet[2] = L1; // L1 Field for fileSize nr of Bytes

    memcpy(&control_packet[3], &fileSize, L1); // V1 for fileSize, will
be stored in little endian order

    control_packet[3+L1] = FILE_NAME; // T2 field for filename

    unsigned int L2 = strlen(filename);
    control_packet[4+L1] = L2; // L2 Field for fileSize nr of Bytes

    memcpy(&control_packet[5+L1], filename, L2); // V2 for fileSize

    return 3 + L1 + 2 + L2; // Size of control Packet
}

int buildDataPacket(unsigned char *data_packet, int control, int
sequenceNR, int buf_size){

    data_packet[0] = control; // Control field

    data_packet[1] = sequenceNR % 255; // Sequence NR mod 255

    data_packet[2] = buf_size/256; // L2

    data_packet[3] = buf_size % 256; // L1

    return 4;
}

void byteStuff(unsigned char *info, int *size, int *bcc2, int bufSize,
unsigned char *buf){

    int i = 0;

    //Byte stuffing and BCC2 calculating
    for(int i = 0; i < bufSize; i++){
        *bcc2 ^= buf[i];
```

```
        if(buf[i] == 0x7e){
            info[*size] = 0x7d;
            info[*size+1] = 0x5e;
            *size += 2;
        }
        else if(buf[i] == 0x7d){
            info[*size] = 0x7d;
            info[*size+1] = 0x5d;
            *size += 2;
        }
        else{
            info[*size] = buf[i];
            *size += 1;
        }
    }

    //BCC2
    if(*bcc2 == 0x7e){
        info[*size] = 0x7d;
        info[*size + 1] = 0x5e;
        info[*size + 2] = FLAG; //Flag
        *size += 3;
    }
    else if(*bcc2 == 0x7d){
        info[*size] = 0x7d;
        info[*size + 1] = 0x5d;
        info[*size + 2] = FLAG; //Flag
        *size += 3;
    }
    else{
        info[*size] = *bcc2;
        info[*size + 1] = FLAG; //Flag
        *size += 2;
    }
}

int byteDestuff(unsigned char *packet, int *size, unsigned char *buf){

    int escape = 0, j = 0;
    *size-=1;
    int counter = 0;

    // Start of packet to BCC2 field
    for(int i = 4; i < *size; i++){

        //byte de-stuffing
```

```
        //0x7d 0x5e -> 0x7e
        //0x7d 0x5d -> 0x7d
        if(buf[i] == 0x7d){
            escape = 1;
            counter++;
            continue;
        }

        if(escape){
            if(buf[i] == 0x5e){
                escape = 0;
                packet[j] = 0x7e;
            }
            else if(buf[i] == 0x5d){
                escape = 0;
                packet[j] = 0x7d;
            }
        }
        else{
            packet[j] = buf[i];
        }

        j++;
    }
    *size-=4;
    *size -= counter;
}

int calculateBCC2(unsigned char *packet, int size){

    int bcc2 = 0;

    for(int i = 0; i < size-1; i++){
        bcc2 ^= packet[i];
    }

    return bcc2;
}
```

application_layer.c

```
// Application layer protocol implementation

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

```c
#include "macros.h"
#include "auxil.h"

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


#define INFO_SIZE MAX_PAYLOAD_SIZE
#define CONTROL_PACKET_SIZE 100

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

    struct timeval tval_before, tval_after, tval_result;

    //Create and setup Link Layer
    LinkLayer Link_Layer;
    strcpy(Link_Layer.serialPort, serialPort);

    if(strcmp(role, "tx")==0) Link_Layer.role = LlTx;
    else if(strcmp(role, "rx")==0) Link_Layer.role = LlRx;

    Link_Layer.baudRate = baudRate;
    Link_Layer.nRetransmissions = nTries;
    Link_Layer.timeout = timeout;

    llopen(Link_Layer);

    switch (Link_Layer.role)
    {
    //Transmiter
    case LlTx:{

        //open file
        FILE *sentFileFd = fopen(filename, "r");

        //check if file exists
        if (sentFileFd==NULL)
        {
            fprintf(stderr,"Invalid file");
            exit(-1);
        }
```

```
        // Get file size in bytes and convert to hex
        fseek(sentFileFd, 0L, SEEK_END);
        unsigned int fileSize = ftell(sentFileFd);
        rewind(sentFileFd);

        //build special start packet
        unsigned char start_control_packet[CONTROL_PACKET_SIZE] = {0};

        int ctrlPacketSize = buildControlPacket(start_control_packet,
START, fileSize, filename);

        //llwrite special start packet
        llwrite(&start_control_packet, ctrlPacketSize);

        //split file into pieces
        unsigned char data_packet[2*INFO_SIZE+4] = {0};
        unsigned char data[2*INFO_SIZE];

        int bytes, dataSize, sequenceNR = 0;

        //while not end of file read from file
        while(bytes = fread(data, 1, INFO_SIZE, sentFileFd)){

            // build data packet
            dataSize = buildDataPacket(data_packet, DATA, sequenceNR,
bytes);
            memcpy(&data_packet[4], data, bytes);

            // llwrite
            if(llwrite(&data_packet,
bytes+dataSize)==-1){fprintf(stderr, "LLwrite error\n"); break;};
            sequenceNR++;
        }


        //build special end packet
        unsigned char end_control_packet[CONTROL_PACKET_SIZE] = {0};

        ctrlPacketSize = buildControlPacket(end_control_packet, END,
fileSize, filename);

        //llwrite special end packet
        llwrite(&end_control_packet, ctrlPacketSize);

        break;
```

```
    }
    //Receiver
    case LlRx:{

        //create and open received file
        FILE *fp;
        fp = fopen (filename, "w");

        unsigned char packet[2*INFO_SIZE], data[INFO_SIZE];

        while(TRUE){

            if(llread(&packet)==-1){
                printf("Error\n");
                continue;
            }

            if(packet[0] == DATA){
                unsigned int sequenceNR = packet[1];
                unsigned int L2 = packet[2];
                unsigned int L1 = packet[3];
                unsigned int L2L1 = (256*L2) + L1;

                //store in file
                fwrite(packet+4, 1, L2L1, fp);
            }
            else if(packet[0] == START){
                fprintf(stderr, "Starting file transmission\n");
                continue;
            }
            else if(packet[0] == END){
                fprintf(stderr, "Ending file transmission\n");
                break;
            }

        }

        break;
    }
    default:
        break;
    }
     llclose(0);
}
```

state_machines.h

```
typedef enum{
    Start,
    FLAG_RCV,
    A_RCV,
    C_RCV,
    BCC_OK,
    STOP
} state_machine;
```

macros.h

```
/*
    Global
*/
#define FLAG 0x7E //delimiter flag
#define AddrTR 0x03 //commands sent by the Transmitter and Replies sent
by the Receiver
#define AddrRT 0x01 //commands sent by the Receiver and Replies sent by
the Transmiter

/*
    Supervision (S) and Unnumbered (U) Frames
*/
#define SET 0x03 //control field = set up
#define DISC 0x0B //control field = disconnect
#define UA 0x07 //control field = unnumbered acknowledgment
#define PosACK(Nr) (Nr<<7)|0x05 //control field = positive
acknowledgement
#define NegACK(Nr) (Nr<<7)|0x01 //control field = negative
acknowledgement

/*
    Information (I) Frames
*/
#define CONTROL(Ns) (Ns<<6) //information frame 0 or 1, is either 0x00
or 0x40

/*
    Packets
*/
#define DATA 0x01 //data control field
#define START 0x02 //start control field
#define END 0x03 //end control field
```

```
#define FILE_SIZE 0x00 //T for file size
#define FILE_NAME 0x01 //T for file name
```