Protocolo de Ligação de Dados

REDES DE COMPUTADORES
LICENCIATURA EM ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES

PEDRO MIGUEL TEIXEIRA VENTURA DA SILVA, UP201905678
TIAGO DA SILVA RIBEIRO, UP201906357

Índice

Sumário

O projeto foi desenvolvido no contexto do primeiro trabalho prático no âmbito da unidade curricular de “Redes de Computadores”, cujo objetivo foi implementar um protocolo de transmissão de dados entre dois computadores distintos com recurso a uma porta série assíncrona, bem como os respetivos testes para comprovar o seu funcionamento e rigidez do código perante os erros, quer introduzidos manualmente, quer inerentes ao trabalho.

O trabalho desenvolvido foi completado com sucesso, respondendo a todos os objetivos propostos, conseguindo-se assim enviar sem perdas de informação diferentes tipos de ficheiros através da porta série.

Introdução

A realização deste trabalho tinha como objeito a implementação de um protocolo funcional de transmissão de dados entre dois computadores, um funcionado como emissor e outro como recetor. Serve o presente relatório como complemento ao trabalho, para documentação e explicação de todo o código utilizado, bem como a explicação de todas as funções criadas de forma a complementar as já fornecidas previamente. Em adição, é ainda apresentado as diversas estatísticas, elementos de valorização e esclarecimento de erros previamente requisitados para o trabalho prático.

O presente relatório está dividido em 6 secções que irão ser abordadas em adiante, sendo elas:

- **Arquitetura –** Explicação dos blocos funcionais e de interface, onde se mostra como compilar e executar corretamente todo o código;
- **Estrutura do Código –** Informação sobre a API, principais estruturas de dados e principais funções criadas e utilizadas no trabalho;
- **Casos de Uso Principais –** Indicação sobre a sequência de chamadas de funções;
- **Protocolo de Ligação Lógica –** Identificação dos principais aspetos funcionais, bem como a descrição da estratégia utilizada para a sua implementação;
- **Validação –** Descrição de testes efetuados para validação da solução apresentada;
- **Conclusão –** Reflexão e síntese da informação apresentada e do trabalho prático realizado e informação sobre os elementos de valorização implementados.

Ir-se-á ainda apresentar, no final, o código fonte do trabalho prático realizado.

Arquitetura

No trabalho realizado, foi necessário ter conhecimento da camada de ligação, bem como da camada de aplicação. No entanto, apenas foi necessária a implementação da camada de ligação, uma vez que a de aplicação foi previamente disponibilizada para a realização do trabalho, que se encontra agregada com a chamada das funções da camada de ligação efetuadas no ficheiro **main.c**. Os ficheiros da camada de ligação são o **linklayer.c** e **linklayer.h**. Além disso, foram criados adicionalmente mais 3 ficheiros: o **data.c**, o **aux.c**, e o **aux.h**. Estes são ficheiros adicionais que auxiliam a implementação da camada de ligação.

Para a compilação do ficheiro foi utilizado um *makefile*, com o seguindo código:

```
CC = gcc
C_FLAGS = -g -Wall

gcc:
    $(CC) $(C_FLAGS) -c linklayer.c aux.c data.c
    $(CC) main.c linklayer.o aux.o data.o -o app

txlab:
    ./app /dev/ttyS0 tx penguin.gif

rxlab:
    ./app /dev/ttyS1 rx penguin-r.gif

clear:
    rm -f app a.out
    rm -f *.o
    rm -f penguin-recieved.gif
```

Desta maneira, para proceder à correta compilação do código, é necessário seguir os seguintes passos:

1. **make clear –** para limpar os ficheiros **.o** e o executável, caso já existam - opcional;
2. **make gcc –** para compilar o **linklayer.c**, **o aux.c** e o **data.c** e compilar de seguida o **main.c** com os ficheiros **.o** resultantes da compilação anterior;
3. **make txlab / make rxlab –** executa o programa, utilizando-se **make txlab** no caso do transmissor e **make rxlab** no caso do **recetor.** Ao executar, é necessário incluir, para além do nome do executável, a porta série que se vai utilizar (**/dev/ttySx**), o modo a utilizar (**tx/rx**, transmissor/recetor respetivamente) e o nome do ficheiro a enviar (**penguin.gif**)

Disclaimer – o número correspondente à porta série pode não coincidir com o utilizado para o trabalho prático.

Estrutura do Código

Como já foi referido, para a implementação da camada de ligação foram criados 3 ficheiros adicionais para auxiliar os ficheiros principais (**linklayer.c** e **linklayer.h**);

- **aux.h –** declaração de todas as funções utilizadas ao longo do trabalho e definições de variáveis;
- **aux.c –** Criação de todas as funções relativas a máquinas de estado e outras de uso esporádico/único;
- **data.c –** Realização das principais funções a utilizar na transmissão da informação.

Assim, estes ficheiros complementam os originais. Nestes, podemos encontrar além das 4 funções principais (**llopen, llwrite, llread, llclose**), algumas outras correspondentes ao alarme, configuração da conexão e estatísticas.

É ainda importante notar a existência do ficheiro **main.c**, disponibilizado previamente que implementa a camada de aplicação e "chama" as 4 principais funções do **linklayer.c**, que tiveram de ser devidamente elaboradas.

Casos de Uso Principais

O principal caso de uso desenvolvido ao longo do trabalho prático é a transferência de um ficheiro entre dois computadores com recurso à porta série, comportando-se um como emissor e outro como recetor. Com efeito, após a compilação e correta execução do programa, indicando todos os parâmetros mandatórios, o programa inicia-se, seguindo uma sequência lógica de transmissão de informação que corresponde ao método ***Stop & Wait***, sendo assim configurada a conexão entre os dois computadores com troca de pacotes para abertura de ligação, seguido do envio de tramas de informação pelo recetor e confirmação da sua receção pelo recetor e por fim, no término do envio de toda a informação, e cessada a ligação entre os dois computadores, trocando protocolos encerramento de conexão.

Protocolo de Ligação Lógica

llopen

Esta função é responsável pelo estabelecimento e configuração da conexão entre os dois computadores. A função recebe apenas um parâmetro, correspondente a uma *struct*, tendo esta vários parâmetros intrínsecos que são necessários ao estabelecimento da ligação. Esta função é utilizada quer pelo transmissor, quer pelo recetor e retorna o valor 1 em caso se sucesso e -1 em caso de insucesso. No entanto, a função toma diferentes finalidades consoante o modo selecionado, recorrendo-se a um *switch* para determinar esse modo:

- **Emissor -** Nesta situação, o programa é responsável por iniciar a ligação, sendo que para isso envia um pacote **SET**, criado através da função **createPkg**, para o recetor. Após o envio, fica à espera de receber um pacote **UA** do recetor. Se receber a resposta antes do tempo necessário para ativar o alarme (**timeOut**), a conexão encontra-se estabelecida e é possível avançar para a próxima fase. Caso não receba a resposta no intervalo de tempo necessário, o programa procede à retransmissão do pacote **SET**, e voltar a ficar à espera de resposta. Se ao fim do número máximo de retransmissões o emissor não receber resposta, a ligação é obrigatoriamente terminada.
- **Recetor –** Neste caso, o programa fica à espera de receber um pacote de início de estabelecimento de ligação. Aquando da receção desse pacote, o recetor é responsável por enviar um pacote **UA**, com recurso à função **createPkg** que indica que recebeu a trama do emissor e está pronto para receber informação.

llwrite

Esta função é responsável pelo envio de tramas de informação com os dados do ficheiro a transmitir. Só o emissor é que executa esta função. Esta recebe dois parâmetros, um correspondente à trama de informação a enviar e outro ao seu tamanho. Esta retorna 0 em caso de sucesso.

Por intermédio da função **createInfoPkg**, é criada uma trama de informação com os dados que se quer enviar para o recetor e com o respetivo tamanho. Nesta função são chamadas outras duas: o **createBCC2**, responsável por criar o BCC2, e o **byte_stuffing**, responsável por fazer o *stuffing* dos dados. Por fim, a função retorna o pacote de dados com *stuffing* criado. É esta a trama de informação enviada para o recetor. Após este envio, o emissor fica à espera da receção de um pacote **RR** correspondente (com o bit de paridade expectável) do recetor que o informa da receção da trama enviada. Se não receber no intervalo de tempo estipulado, com recurso à função **timeOut**, procede à sua retransmissão até um número máximo de retransmissões. Caso exista ainda um pedido de **REJ**, o emissor procede à retransmissão da trama mais rapidamente, sem ter que esperar pelo alarme. No final da função, é ainda atualizado os bits de paridade da trama enviada, segundo a norma estipulada e ainda libertado o espaço de memória gerado para alocar o pacote enviado, com recurso ao comando *free*.

- ## llread

Esta função é responsável pela leitura e tratamento das tramas de dados previamente enviadas pelo emissor. Só o recetor é que executa esta função. Esta apenas recebe um parâmetro que corresponde ao pacote resultante da leitura e tratamento da trama de informação enviada pelo emissor, retornando assim o número de *bytes* resultantes.

Com efeito, na implementação feita, a função ocupa-se de ler *byte* a *byte* e, após a leitura do cabeçalho, coloca num pacote apenas os dados lidos, ou seja, retirando o H*eader*. Após este processo, o programa está nas condições de assumir que a penúltima posição corresponde ao BCC2 enviado pelo emissor e assim, faz o *destuffing* de todos os dados recebidos, atribuindo-os ao *packet*, por intermédio da função **byte_destuffing**, que depois retorna o número de *bytes* do mesmo. Para efeito de descoberta de erros, utiliza-se novamente a função **createBCC2** para descobrir o BCC2 do *packet*. Por conseguinte, compara-se ambos os BCC2's. Caso não sejam iguais, está-se na condição de assumir que ocorreu um erro na transmissão dessa trama e assim emite-se um **REJ** que se envia para o emissor. Caso os BCC2's coincidam, então a trama enviada está efetivamente correta, enviando-se um pacote **RR** que assinala a receção correta da informação e ainda atualiza-se o bit de paridade.

- ## llclose

Esta função é responsável por cessar a conexão entre o transmissor e o recetor. A função apenas recebe um parâmetro relativo a disponibilização de estatísticas. Ambos o emissor e recetor utilizam esta função que retorna 0 em caso de sucesso. De modo análogo ao **llopen**, também esta função tem diferentes finalidades consoante o modo em que se utiliza. Assim, foi usado um *switch* para o determinar.

- **Emissor –** Nesta situação, o programa é responsável por terminar a ligação. Com esta finalidade, envia um pacote **DISC**, por intermédio da função **createPkg**. Após o envio, fica à espera de uma resposta positiva do recetor, novamente com um pacote **DISC**. Caso não aconteça no intervalo de tempo estipulado, com recurso à função **timeOut**, procede à sua retransmissão até um número máximo de retransmissões. Após a confirmação do recetor, o emissor envia ainda um pacote **UA**, novamente com recurso à função **createPkg**, terminando assim a conexão e a transmissão de toda e qualquer informação entre os computadores.
- **Recetor –** Nesta situação, o programa fica à espera de um pacote de término da conexão enviado pelo transmissor. Aquando da sua receção, o recetor, utilizando a função **createPkg**, envia um pacote **DISC** para o emissor, informando-o assim que recebeu a sua informação para cessar a ligação.

Validação

Com o objetivo de validar e comprovar a rigidez da solução encontrada para implementar o protocolo de transmissão de dados, foram realizados diversos testes:

1. Envio de ficheiros com vários formatos e diferentes tamanhos (para submissão apenas se considera o formato .png para além do disponibilizado);
2. Interrupção manual e digital da ligação e consequente transmissão da informação entre os dois computadores;
3. Envio de ficheiros com diversos valores de baudrate;
4. Envio de ficheiros com informação a ser transmitida com ruído;
5. Envio de ficheiros com diminuição do tamanho máximo da trama de informação.

É importante referir que a solução implementada passou a todos os testes efetuados com sucesso, conseguindo-se assim enviar e receber corretamente o ficheiro pretendido através da porta série.

Conclusão

Através da implementação com sucesso de um protocolo de ligação e transmissão de dados, conseguiu-se perceber a complexidade do processo, entender o seu funcionamento e compreender a necessidade da existência de critérios de identificação de erros e consequente retransmissão de tramas defeituosas. Foi também possível validar a independência das camadas de ligação e aplicação, uma vez que a camada de ligação oferecia serviços à camada de aplicação, desconhecendo os detalhes da sua implementação, sendo assim independentes uma da outra.

Foram ainda cumpridos todos os elementos de valorização propostos, sendo também todos implementados com sucesso, como é possível comprovar no código fonte abaixo disponibilizado.

Por último, é mais uma vez importante referir que o objetivo do trabalho foi cumprido com sucesso e pode-se depreender ainda que o trabalho prático realizado funcionou como um complemento à matéria teórica lecionada na unidade curricular de “Redes de Computadores”.

Anexos – Código Fonte

- **makefile**

```
CC = gcc
C_FLAGS = -g -Wall

gcc:
    $(CC) $(C_FLAGS) -c linklayer.c aux.c data.c
    $(CC) main.c linklayer.o aux.o data.o -o app

tx:
    ./app /dev/ttyS10 tx penguin.gif

rx:
    ./app /dev/ttyS11 rx penguin-r.gif

txlab:
    ./app /dev/ttyS0 tx penguin.gif

rxlab:
    ./app /dev/ttyS1 rx penguin-r.gif

tx_png:
    ./app /dev/ttyS10 tx rubiks_cube.png

rx_png:
    ./app /dev/ttyS11 rx rubiks_cube_r.png

clear:
    rm -f app
    rm -f *.o
    rm -f penguin-r.gif rubiks_cube_r.png
```

- **aux.h**

```
#ifndef AUX
#define AUX

//used for debugging
//#include <time.h>

#define FLAG 0x7E
#define ESC 0x7d
#define STUFF 0x20
#define A 0x03
#define A_2 0x01

#define C_SET       0x03
#define C_UA        0x07
#define C_DISC      0x0b

#define C_I(x)      (0x00 | (x << 1))
#define C_RR(x)     (0x01 | (x << 5))
#define C_REJ(x)    (0x05 | (x << 5))

#define BCC_SET     (A^C_SET)
#define BCC_UA      (A^C_UA)
#define BCC_DISC    (A^C_DISC)

#define START_STATE 0
#define FLAG_STATE  1
#define A_STATE     2
#define C_STATE     3
#define BCC_STATE   4
#define STOP_STATE  5
#define DATA_STATE  6
#define C_REJ_STATE 7
#define BCC_REJ_STATE 8
#define STOP_REJ_STATE 9

/* These defines are meant to be used with the createPkg() function */
#define SET_pkg 1
#define UA_pkg 2
#define UA2_pkg 4
#define DISC_pkg 3
#define RR_pkg 5
#define REJ_pkg 6

/* Bits de paridade */
extern int parity_bit;

typedef struct stats
```

```
{
    int numBytesStuff;
    int numBytesFile;
    int numREJ;
    int numTimeouts;
    int numIframes;
    int numRetransmissions;
} varStatistics;

extern varStatistics stats;

/*
 * Cada State Machine permite detetar os diferentes tipos de tramas,
respetivamente
 *
 * Para usar este tipo de funcoes, devera ser fornecido, atraves da
variavel tx, o
 * byte que esta a ser lido nesse momento, bem como o estado atual
 */
int StateMachineUA(unsigned char tx, int state);
int StateMachineDISC(unsigned char tx, int state);
int StateMachineSET(unsigned char tx, int state);
int StateMachineUA2(unsigned char tx, int state);
int StateMachineRR_REJ(unsigned char tx, int state);
int StateMachineI(unsigned char tx, int state);

/*
 * Escreve em pkg o tipo de pacote especificado em "type" (nao
aplicavel
 * para a trama de informacao)
 *
 * No caso dos bits de paridade, esta funcao limita-se apenas a
utilizar
 * o valor que da variavel global "parity_bit" em vigor nesse momento,
sendo
 * necessario defini-lo corretamente antes de efetucar a sua chamada
 */
void createPkg(unsigned int type, unsigned char * pkg);

/* Cria a trama de Informacao
 *
 * Recebe os dados antes do Stuffing e o seu respetivo tamanho
 * Aloca o tamannho estritamente necessario para a trama apos o
Stuffing
 * Chama a funcao byte_stuffing
 * Chama a funcao createBCC2
 * Devolve na variavel pkg a trama I completa
 */
```

```
unsigned char * createInfoPkg(unsigned char * data, int sizeData, int
* finalSize);
int byte_stuffing(unsigned char *buf, int bufSize, unsigned char
*stuffedBuf);
unsigned char createBCC2(unsigned char *data, int lenght);

int byte_destuffing(unsigned char *StuffedBuf, int StuffSize, unsigned
char *deStuffedBuf);

/* utility functions */
int getBaud(int baud);
void timeOut();
void connectionConfig(linkLayer connectionParameters);
void llcopy(linkLayer connectionParameters);
void printstatistics();

/* For debugging only */
void printFlags(unsigned char x);
void printInfoPkg(int size, unsigned char *pkg, unsigned char BCC2);
#endif
```

- **aux.c**

```c
#include "linklayer.h"
#include "aux.h"

int getBaud(int baud)
{
    switch (baud)
    {
    case 9600:
        return B9600;
    case 19200:
        return B19200;
    case 38400:
        return B38400;
    case 57600:
        return B57600;
    case 115200:
        return B115200;
    case 230400:
        return B230400;
    case 460800:
        return B460800;
    case 500000:
        return B500000;
    case 576000:
        return B576000;
    case 921600:
        return B921600;
    case 1000000:
        return B1000000;
    case 1152000:
        return B1152000;
    case 1500000:
        return B1500000;
    case 2000000:
        return B2000000;
    case 2500000:
        return B2500000;
    case 3000000:
        return B3000000;
    case 3500000:
        return B3500000;
    case 4000000:
        return B4000000;
    default:

        return -1;
    }
```

```
}

void createPkg(unsigned int type, unsigned char * pkg)
{
    switch (type)
    {
    case SET_pkg:
        pkg[0] = FLAG;
        pkg[1] = A;
        pkg[2] = C_SET;
        pkg[3] = BCC_SET;
        pkg[4] = FLAG;
        break;

    case UA_pkg:
        pkg[0] = FLAG;
        pkg[1] = A;
        pkg[2] = C_UA;
        pkg[3] = BCC_UA;
        pkg[4] = FLAG;
        break;

    case UA2_pkg:
        pkg[0] = FLAG;
        pkg[1] = A_2;
        pkg[2] = C_UA;
        pkg[3] = A_2 ^ C_UA;
        pkg[4] = FLAG;
        break;

    case DISC_pkg:
        pkg[0] = FLAG;
        pkg[1] = A;
        pkg[2] = C_DISC;
        pkg[3] = BCC_DISC;
        pkg[4] = FLAG;
        break;
    case RR_pkg:
        pkg[0] = FLAG;
        pkg[1] = A;
        pkg[2] = C_RR(parity_bit);
        pkg[3] = A ^ pkg[2];
        pkg[4] = FLAG;
        break;
    case REJ_pkg:
        pkg[0] = FLAG;
        pkg[1] = A;
        pkg[2] = C_REJ(parity_bit);
        pkg[3] = A ^ pkg[2];
```

```
        pkg[4] = FLAG;
        break;
    }
}

int StateMachineSET(unsigned char tx, int state)
{
    switch (state)
    {
    case START_STATE:
        if (tx == FLAG)
            state = FLAG_STATE;
        break;

    case FLAG_STATE:
        if (tx == A)
            state = A_STATE;
        else if (tx == FLAG)
            state = FLAG_STATE;
        else state = START_STATE;
        break;

    case A_STATE:
        if (tx == C_SET)
            state = C_STATE;
        else if (tx == FLAG)
            state = FLAG_STATE;
        else state = START_STATE;
        break;

    case C_STATE:
        if (tx == BCC_SET)
            state = BCC_STATE;
        else if (tx == FLAG)
            state = FLAG_STATE;
        else state = START_STATE;
        break;

    case BCC_STATE:
        if (tx == FLAG)
            state = STOP_STATE;
        else
            state = START_STATE;
        break;

    case STOP_STATE:
        break;
    }
```

```
    return state;
}

int StateMachineUA(unsigned char tx, int state)
{
    switch (state)
    {
    case START_STATE:
        if (tx == FLAG)
            state = FLAG_STATE;
        break;

    case FLAG_STATE:
        if (tx == A)
            state = A_STATE;
        else if (tx == FLAG)
            state = FLAG_STATE;
        else state = START_STATE;
        break;

    case A_STATE:
        if (tx == C_UA)
            state = C_STATE;
        else if (tx == FLAG)
            state = FLAG_STATE;
        else state = START_STATE;
        break;

    case C_STATE:
        if (tx == BCC_UA)
            state = BCC_STATE;

        else if (tx == FLAG)
            state = FLAG_STATE;

        else
            state = START_STATE;
        break;

    case BCC_STATE:
        if (tx == FLAG)
            state = STOP_STATE;

        else
            state = START_STATE;

        break;

    case STOP_STATE:
```

```
        break;
    }

    return state;
}

int StateMachineUA2(unsigned char tx, int state)
{
    switch (state)
    {
    case START_STATE:
        if (tx == FLAG)
            state = FLAG_STATE;
        break;

    case FLAG_STATE:
        if (tx == A_2)
            state = A_STATE;
        else if (tx == FLAG)
            state = FLAG_STATE;
        else state = START_STATE;
        break;

    case A_STATE:
        if (tx == C_UA)
            state = C_STATE;
        else if (tx == FLAG)
            state = FLAG_STATE;
        else state = START_STATE;
        break;

    case C_STATE:
        if (tx == (A_2 ^ C_UA))
            state = BCC_STATE;
        else if (tx == FLAG)
            state = FLAG_STATE;
        else state = START_STATE;
        break;

    case BCC_STATE:
        if (tx == FLAG)
            state = STOP_STATE;
        else
            state = START_STATE;
        break;

    case STOP_STATE:
        break;
    }
```

```
    return state;
}

int StateMachineDISC(unsigned char tx, int state)
{
    switch (state)
    {
    case START_STATE:
        if (tx == FLAG)
            state = FLAG_STATE;

        break;

    case FLAG_STATE:
        if (tx == A)
            state = A_STATE;

        else if (tx == FLAG)
            state = FLAG_STATE;

        else
            state = START_STATE;
        break;

    case A_STATE:
        if (tx == C_DISC)
            state = C_STATE;

        else if (tx == FLAG)
            state = FLAG_STATE;

        else
            state = START_STATE;

        break;

    case C_STATE:
        if (tx == BCC_DISC)
            state = BCC_STATE;

        else if (tx == FLAG)

            state = FLAG_STATE;

        else
            state = START_STATE;
        break;
```

```
    case BCC_STATE:
        if (tx == FLAG)
            state = STOP_STATE;

        else
            state = START_STATE;

        break;

    case STOP_STATE:
        break;
    }

    return state;
}

int StateMachineI(unsigned char tx, int state)
{
    switch (state)
    {
    case START_STATE:
        if (tx == FLAG)
            state = FLAG_STATE;

        break;

    case FLAG_STATE:
        if (tx == A)
            state = A_STATE;

        else if (tx == FLAG)
            state = FLAG_STATE;

        else
            state = START_STATE;
        break;

    case A_STATE:
        if (tx == C_I(parity_bit))
            state = C_STATE;

        else if (tx == FLAG)
            state = FLAG_STATE;

        else
            state = START_STATE;

        break;
```

```
    case C_STATE:
        if (tx == (A ^ C_I(parity_bit)))
            state = BCC_STATE;

        else if (tx == FLAG)
            state = FLAG_STATE;

        else
            state = START_STATE;
        break;

    case BCC_STATE:
        if (tx == FLAG)
            state = FLAG_STATE;

        else
            state = DATA_STATE;

        break;

    case DATA_STATE:
        if (tx == FLAG)
            state = STOP_STATE;

        break;
    case STOP_STATE:
        break;

    }

    return state;
}

int StateMachineRR_REJ(unsigned char tx, int state)
{
    switch (state)
    {
    case START_STATE:
        if (tx == FLAG)
            state = FLAG_STATE;

        break;

    case FLAG_STATE:
        if (tx == A)
            state = A_STATE;

        else if (tx == FLAG)
            state = FLAG_STATE;
```

```
        else
            state = START_STATE;
        break;

    case A_STATE:
        if (tx == C_RR((1-parity_bit)))
            state = C_STATE;

        else if (tx == C_REJ((1-parity_bit)))
            state = C_REJ_STATE;

        else if (tx == FLAG)
            state = FLAG_STATE;

        else
            state = START_STATE;

        break;


    /*********************RR frame***********************/
    case C_STATE:
        if (tx == (A ^ C_RR((1-parity_bit))))
            state = BCC_STATE;


        else if (tx == FLAG)
            state = FLAG_STATE;

        else
            state = START_STATE;
        break;

    case BCC_STATE:
        if (tx == FLAG)
            state = STOP_STATE;

        else
            state = START_STATE;

        break;

    case STOP_STATE:
        break;

    /*********************REJ frame***********************/
    case C_REJ_STATE:
        if (tx == (A ^ C_REJ((1-parity_bit))))
```

```
            state = BCC_REJ_STATE;


        else if (tx == FLAG)
            state = FLAG_STATE;

        else
            state = START_STATE;
        break;

    case BCC_REJ_STATE:
        if (tx == FLAG)
            state = STOP_REJ_STATE;

        else
            state = START_STATE;

        break;

    case STOP_REJ_STATE:
        break;

    }

    return state;
}


/* for debugging only */
void printFlags(unsigned char x)
{
    switch (x)
    {
    case FLAG:
        printf("FLAG");
        break;

    default:
        printf("%x", x);
        break;
    }
}

void printInfoPkg(int size, unsigned char *pkg, unsigned char BCC2)
{
    for(int i = 0; i < size; i++)
    {
        if(pkg[i] == FLAG)  printf("[i:%d FLAG] ", i);
```

```
        else if(pkg[i] == A)      printf("[i:%d A] ", i);

        else if(pkg[i] == C_I(0) || pkg[i] == C_I(1)
)    printf("[i:%d C] ", i);

        else if(pkg[i] == (A^C_I(0)) || pkg[i] == (A^C_I(1)) )
printf("[i:%d BCC1] ", i);

        else if(pkg[i] == BCC2) printf("[i:%d BCC2] ", i);

        else printf("[%u]", pkg[i]);
    }
    printf("\n");
}
```

- **data.c**

```
#include "linklayer.h"
#include "aux.h"

unsigned char * createInfoPkg(unsigned char * data, int sizeData, int*
finalSize)
{
    if (!data || sizeData > MAX_PAYLOAD_SIZE || !finalSize)
    {
        puts("Couldnt createInfoPkg");
        exit(-1);
    }

    int extraSize = 0;
    unsigned char * pkg;


    for(int i = 0; i < sizeData; i++)
    {
        if(data[i] == FLAG || data[i] == ESC)
            extraSize++;
    }

    *finalSize = 6 + sizeData + extraSize;
    pkg = calloc(sizeof(unsigned char), *finalSize);

    pkg[0] = FLAG;
    pkg[1] = A;
    pkg[2] = C_I(parity_bit);
    pkg[3] = (A ^ pkg[2]);
    pkg[*finalSize - 2] = createBCC2(data, sizeData);
    pkg[*finalSize - 1] = FLAG;

    byte_stuffing(data, sizeData, (pkg + 4));

    stats.numBytesStuff += extraSize;

    /*  used to virtually generate errors, and test REJ

    unsigned char val;
    int randPos[10];
    for(int i = 0; i < 10; i++)
    {
        randPos[i] = rand() % sizeData;
        if(randPos[i] < 4) randPos[i] += 4;
        val = rand();
        pkg[randPos[i]] = val;
        printf("%d: Pkg[%d] = %u\n", i, randPos[i], val);
```

```
    }
    */

    return pkg;
}

int byte_stuffing(unsigned char *buf, int bufSize, unsigned char
*stuffedBuf)
{
    if (!buf || bufSize < 0 || !stuffedBuf || bufSize >
MAX_PAYLOAD_SIZE)
    {
        puts("Bad parameters in byte_stuffing()");
        exit(-1);
    }

    int i, j = 0;

    for (i=0; i< bufSize; i++)
    {
        if ((buf[i] == FLAG))
        {
            stuffedBuf[j] = ESC;
            stuffedBuf[j+1] = (FLAG^STUFF);
            j += 2;
        }
        else if (buf[i] == ESC)
        {
            stuffedBuf[j] = ESC;
            stuffedBuf[j+1] = (ESC^STUFF);
            j += 2;
        }
        else
        {
            stuffedBuf[j] = buf[i];
            j++;
        }

    }

    return j;
}

int byte_destuffing(unsigned char *stuffedBuf, int StuffSize, unsigned
char *buf)
{
    if (!stuffedBuf || !buf || StuffSize > 2*MAX_PAYLOAD_SIZE)
    {
        puts("Bad parameters destuffing");
```

```
        exit(-1);
    }
    int i, newpos;
    for (i=0, newpos = 0; i < StuffSize; i++, newpos++)
    {
        if (stuffedBuf[i] == ESC)
        {
            if (stuffedBuf[i+1] == (FLAG^STUFF)){
                buf[newpos] = FLAG;
                i++;
            }
            else if (stuffedBuf[i+1] == (ESC^STUFF)){
                buf[newpos] = ESC;
                i++;
            }
        }
        else buf[newpos] = stuffedBuf[i];
    }
    return newpos;
}

unsigned char createBCC2(unsigned char *data, int lenght)
{
    if (!data || lenght < 0)
    {
        puts("Couldnt create BCC");
        exit(-1);
    }

    unsigned char buf = 0x00;
    for (int i=0; i<lenght; i++)
        buf ^= data[i];

    return buf;
}
```

- **linklayer.c**

```
#include "linklayer.h"
#include "aux.h"

int attempts=1, timeOutFLAG=1;
int fd;
struct termios oldtio,newtio;
linkLayer cP;
int parity_bit = 0;

varStatistics stats;

void timeOut()
{
    printf("Alarm #%d\n", attempts);
    timeOutFLAG=1;
    attempts++;
}

int llopen(linkLayer connectionParameters)
{
    unsigned char b;
    unsigned char buf[5];
    int state = START_STATE;
    ssize_t res = 0, res_old = 0;

    llcopy(connectionParameters);
    connectionConfig(connectionParameters);

    switch (cP.role)
    {
    case TRANSMITTER:
        createPkg(SET_pkg, buf);

        (void) signal(SIGALRM, timeOut);
        stats.numTimeouts--;

        while(attempts <= cP.numTries && state != STOP_STATE){

            res += read(fd, &b, 1);

            if(res == res_old && timeOutFLAG)
            {
                write(fd, buf, 5);
                alarm(cP.timeOut);
                timeOutFLAG = 0;

                stats.numTimeouts++;
```

```
            }

            if(res > res_old)
            {
                alarm(0);
                state = StateMachineUA(b, state);
                timeOutFLAG = 1;
            }

            res_old = res;
        }

        if(attempts > cP.numTries){
            puts("Number of tries exceded");
            exit(-1);
        }

        break;

    case RECEIVER:

        while(state != STOP_STATE)
        {
            read(fd, &b, 1);
            state = StateMachineSET(b, state);
        }

        createPkg(UA_pkg, buf);
        write(fd, buf, 5);
        break;

    default:
        exit(-1);
        break;
    }

    sleep(1);
    return 1;
}

int llwrite(char* buf, int bufSize)
{

    if(buf == NULL || bufSize > MAX_PAYLOAD_SIZE)
        exit(-1);

    unsigned char * pkg = NULL;
    unsigned char b;
    int size, state = START_STATE;
```

```
    ssize_t res = 0, res_old = 0;

    pkg = createInfoPkg((unsigned char *)buf, bufSize, &size);

    stats.numTimeouts--;
    (void) signal(SIGALRM, timeOut);

    attempts = timeOutFLAG = 1;

    while(attempts <= cP.numTries && state != STOP_STATE)
    {
        res += read(fd, &b, 1);

        if(state == STOP_REJ_STATE)
        {
            alarm(0);
            write(fd, pkg, size);
            state = START_STATE;

            stats.numIframes++;
            stats.numREJ++;

            puts("Retransmiting...");
        }

        else if(res > res_old)
        {
            alarm(0);
            state = StateMachineRR_REJ(b, state);
            timeOutFLAG = 1;
        }

        else if(res == res_old && timeOutFLAG)
        {
            write(fd, pkg, size);
            alarm(cP.timeOut);
            timeOutFLAG = 0;

            stats.numTimeouts++;
            stats.numIframes++;
        }

        res_old = res;
    }


    if(attempts > cP.numTries)
    {
        puts("Number of tries exceded");
```

```
        free(pkg);
        exit(-1);
    }

    parity_bit = 1 - parity_bit;
    free(pkg);
    return 0;
}

int llread(char* packet)
{
    unsigned char buf[5], b, pkgRecieved[MAX_PAYLOAD_SIZE * 2];
    int state = START_STATE;
    int i = 0, pkgSize, deStuffSize, BCC2, BCC2_r;
    ssize_t res = 0, res_old = 0;

    while(state != STOP_STATE && i < MAX_PAYLOAD_SIZE * 2)
    {
        res += read(fd, &b, 1);

        if(res > res_old)
        {
            state = StateMachineI(b, state);

            if(state == DATA_STATE)
            {
                pkgRecieved[i] = b;
                i++;
            }
        }

        res_old = res;
    }

    BCC2 = pkgRecieved[i-1];
    pkgSize = i - 1;

    deStuffSize = byte_destuffing(pkgRecieved, pkgSize, (unsigned char
*) packet);
    BCC2_r = createBCC2((unsigned char *) packet, deStuffSize);

    stats.numBytesFile += deStuffSize;

    if(BCC2 != BCC2_r)
    {
        createPkg(REJ_pkg, buf);
        write(fd, buf, 5);

        puts("Package rejected");
```

```
        return 0;
    }

    else
    {
        parity_bit = 1 - parity_bit;
        createPkg(RR_pkg, buf);
        write(fd, buf, 5);
    }

    return deStuffSize;
}

int llclose(int showStatistics)
{
    unsigned char buf[5], b;
    int state = START_STATE;
    ssize_t res = 0, res_old = 0;

    switch (cP.role)
    {
    case TRANSMITTER:
        createPkg(DISC_pkg, buf);

        (void) signal(SIGALRM, timeOut);
        attempts = timeOutFLAG = 1;
        stats.numTimeouts--;

        while(attempts <= cP.numTries && state != STOP_STATE){

            res += read(fd, &b, 1);

            if(res == res_old && timeOutFLAG)
            {
                write(fd, buf, 5);
                alarm(cP.timeOut);
                timeOutFLAG = 0;

                stats.numTimeouts++;
            }

            if(res > res_old)
            {
                alarm(0);
                state = StateMachineDISC(b, state);
                timeOutFLAG = 1;
            }

            res_old = res;
```

```
        }

        if(attempts > cP.numTries){
            puts("Number of tries exceded");
            exit(-1);
        }

        createPkg(UA2_pkg, buf);
        write(fd, buf, 5);
    break;

    case RECEIVER:
        while(state != STOP_STATE)
        {
            res += read(fd, &b, 1);

            if(res > res_old)
                state = StateMachineDISC(b, state);

            res_old = res;
        }

        createPkg(DISC_pkg, buf);
        write(fd, buf, 5);

        state = START_STATE;
        while(state != STOP_STATE)
        {
            res += read(fd, &b, 1);

            if(res > res_old)
                state = StateMachineUA2(b, state);

            res_old = res;
        }
        puts("Everything went smooth");
    break;

    default:
        exit(-1);
        break;
    }

    close(fd);

    if(showStatistics)
        printstatistics ();

    sleep(1);
```

```
    return 0;
}

void llcopy(linkLayer connectionParameters)
{
    cP.role = connectionParameters.role;
    cP.timeOut = connectionParameters.timeOut;
    cP.numTries = connectionParameters.numTries;
}

void connectionConfig(linkLayer connectionParameters)
{
    fd = open(connectionParameters.serialPort, O_RDWR | O_NOCTTY );
    if (fd <0)
    {
        perror(connectionParameters.serialPort);
        exit(-1);
    }

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

    bzero(&newtio, sizeof(newtio));
    newtio.c_cflag = getBaud(connectionParameters.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]    = connectionParameters.timeOut * 10;   /*
inter-character timer unused */
    newtio.c_cc[VMIN]     = 0;   /* blocking read until 1 chars
received */

    tcflush(fd, TCIOFLUSH);

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

```
void printstatistics ()
{
    printf("\nStatistics: \n");

    printf("Number of Bytes of the file stuffed: %d B\n",
stats.numBytesStuff);
    printf("Number of Bytes of the file recieved: %d B\n",
stats.numBytesFile);
    printf("Number of REJ frames: %d\n", stats.numREJ);
    printf("Number of I frames sent: %d\n", stats.numIframes);
    printf("Number of Timeouts: %d\n", stats.numTimeouts);
    stats.numRetransmissions = stats.numREJ + stats.numTimeouts;
    printf("Number of Retransmissions: %d\n",
stats.numRetransmissions);
}
```