Protocolo de Ligação de Dados

1º Trabalho Laboratorial

Redes de Computadores - Turma 7

João Filipe de Menezes Falcão e Sousa Guedes (up202108711@up.pt)

Eduardo Afonso Soares Ferreira Machado (up202105337@up.pt)

Porto, 27 de Outubro de 2023

Índice

Sumário

Este projeto foi realizado no âmbito da unidade curricular de Redes de Computadores e tem como objetivo implementar um protocolo de comunicação de dados onde os ficheiros enviados e recebidos fazem uso da Porta Série Rs-232. Graças a este projeto conseguimos aplicar a matéria lecionada em relação ao protocolo de ligação de dados assim como o funcionamento da estratégia *Stop-and-Wait*.

Introdução

O projeto foi desenvolvido com o objetivo da criação de um protocolo de ligação de dados para a transferência de um ficheiro armazenado no disco de um PC para outro, estando ligados através de um cabo série. O relatório está dividido nas seguintes partes:

- **Arquitetura**: blocos funcionais e interfaces;
- **Estrutura de código**: APIs, principais estruturas de dados, principais funções e a sua relação com a arquitetura;
- **Casos de uso principais**: identificação; sequências de chamada de funções;
- **Protocolo de ligação lógica**: identificação dos principais aspetos funcionais; descrição da estratégia de implementação destes aspetos com apresentação de extratos de código;
- **Protocolo de aplicação:** identificação dos principais aspetos funcionais; descrição da estratégia de implementação destes aspetos com apresentação de extratos de código;
- **Validação:** descrição dos testes efetuados com apresentação quantificada dos resultados, se possível;
- **Eficiência do protocolo de ligação de dados:** caracterização estatística da eficiência do protocolo, efetuada recorrendo a medidas sobre o código desenvolvido;
- **Conclusões:** síntese da informação apresentada nas seções anteriores; reflexão sobre os objetivos de aprendizagem alcançados;

Arquitetura

O nosso projeto foi desenvolvido na base de duas camadas independentes uma da outra, *ApplicationLayer* e *LinkLayer*. A *ApplicationLayer* é a camada de aplicação, responsável pela interação com o utilizador e com o ficheiro a ser enviado/transmitido, cria os pacotes de controlo e informação, e faz uso das funções principais do *LinkLayer*. A *LinkLayer* é a camada de ligação de dados, estabelece e termina a ligação entre as portas série, cria e envia as tramas de dados, encarrega-se de validar as tramas recebidas e de enviar mensagens em caso de erro.

Estrutura do código

O código está dividido em dois ficheiros, correspondentes às duas camadas de arquitetura, *application_layer.c* e *link_layer.c*.

O **application_layer.c** inclui várias funções que desempenham papéis específicos em uma aplicação de transmissão de dados por meio de uma conexão serial. Isso inclui funções para determinar o tamanho de um arquivo, criar pacotes de controlo e dados (*createControlPacket, createDataPacket*), bem como ler e analisar esses pacotes (*readControlPacket, readDataPacket*). A função principal (*applicationLayer*) configura os parâmetros da camada de link, abre a porta e estabelece a conexão. Dependendo do papel especificado (transmissor ou receptor), a função realiza a lógica de transmissão ou recepção, lendo e escrevendo dados de arquivos e controlando a inicialização e encerramento da conexão.

O **link_layer.c** implementa as funcionalidades da camada link de um protocolo de comunicação serial, abrangendo a abertura, transmissão, recepção e fecho de ligações. Utiliza uma máquina de estados para interpretar os bytes recebidos, reconhecer padrões e gerar respostas apropriadas. A função *llopen* estabelece a ligação; *llwrite* envia pacotes de dados com suporte para retransmissão; *llread* lê e valida pacotes de dados; *llclose* encerra a ligação.

Existem header files (application_layer.h; link_layer.h), que contêm declarações importantes, nomeadamente para valores de flags e estados, usadas para efeitos de organização dos dados.

Casos de uso principais

O programa é compilado utilizando o ficheiro Makefile com o comando *make* seguido de run_tx, para o computador que vai servir de transmissor, ou run_rx, para o computador receptor. Para o programa correr corretamente, o programa receptor deverá ser iniciado no começo, enquanto espera pelo transmissor ser iniciado. No momento em que a ligação é estabelecida, o emissor envia os dados ao receptor, que os guardará sob o nome especificado pelo utilizador. Durante o decorrer do programa, mensagens aparecem na consola descrevendo o decorrer da transmissão. Ao ser tudo recebido, verificado e aceite, o programa termina. No caso do transmissor ser iniciado em primeiro e exceder o número de retransmissões permitidas enquanto espera pela resposta do receptor, o programa é terminado, mensagens de erro serão enviadas.

Protocolo de ligação lógica

O protocolo de ligação lógica configura as portas séries, estabelece a ligação entre elas, transfere os dados através dos ports, e recupera potenciais erros que podem ocorrer durante a transferência de dados. Estas ações são (Unnumbered Acknowledgment) efetuadas em quatro funções.

A função **llopen** é responsável por estabelecer a conexão de comunicação entre as partes, atuando como o ponto de partida para a comunicação de dados. Ela envia uma trama SET para iniciar a conexão através da porta série. Após o envio, aguarda uma resposta do receptor, normalmente na forma de um UA (Unnumbered Acknowledgment), que confirma a configuração da conexão. Além disso, essa função realiza a configuração dos parâmetros da porta série, garantindo que a comunicação ocorra de maneira adequada, seguindo os padrões estabelecidos.

A função **llwrite** é essencial na transmissão de dados da application layer para o link layer. Ela empacota os dados em frames, incluindo informações de controlo e verificação de erros para garantir uma transmissão segura e confiável. Além de enviar os dados, a função lida com as respostas do receptor, ao gerar RR quando os dados são recebidos com sucesso ou REJ (Reject) em caso de detecção de erros nos frames. Isso garante que os dados sejam entregues corretamente e permite a retransmissão de pacotes com problemas.

A função **llread** é encarregada de receber e interpretar os frames de dados enviados pelo transmissor. Ela verifica a integridade dos dados, detectando erros que possam ter ocorrido durante a transmissão. Ao receber os dados, a função gera respostas, como RR ou REJ, para informar ao transmissor se os pacotes de dados foram aceites ou rejeitados. Isso é fundamental para garantir a confiabilidade da comunicação, uma vez que permite a correção de erros ou a retransmissão de dados, se necessário.

A função **llclose** é responsável por encerrar a conexão de comunicação entre as partes. Para fazer isso, ela inicia o processo de encerramento enviando um comando DISC (Disconnect) e aguardando uma resposta do receptor. Após receber um DISC como confirmação, a função envia um último UA para finalizar a conexão de forma segura. Além disso, a função restaura os parâmetros da porta série para o estado original, encerrando efetivamente a comunicação entre as partes. Esse processo garante um encerramento adequado da conexão e a disponibilidade da porta série para futuras comunicações.

Protocolo de aplicação

O protocolo de aplicação é responsável pela criação e transferência dos pacotes de controlo e dados; leitura e escrita do ficheiro a transferir.

A função **createControlPacket** tem a responsabilidade de criar um pacote de controlo a ser enviado durante a comunicação de dados. Dependendo do tipo especificado (0 para início e 1 para fim), ela preenche o buffer `buf` com as informações necessárias. Isso inclui a marcação do tipo de pacote de controlo, o tamanho do arquivo em bytes e o seu respectivo nome. O pacote resultante é usado para iniciar ou encerrar uma transferência de arquivo, sendo essencial para a coordenação entre o transmissor (TX) e o receptor (RX).

A função **createDataPacket** é responsável por construir um pacote de dados para transmissão, populando o buffer fornecido `buf` com informações e dados necessários. A estrutura do pacote de dados inclui o tipo de pacote (0x01 para pacotes de dados), o número de sequência, o tamanho dos dados (representação big-endian) e, a partir do byte 4 em diante, o conteúdo de dados real. Esta função é essencial para dividir os dados do arquivo em pacotes menores, permitindo a transmissão e a reconstrução dos arquivos no receptor (RX).

A função **readControlPacket** é responsável por ler e analisar um pacote de controle armazenado no buffer de entrada *buf*. Ela extrai informações, como o tamanho do arquivo e o nome do arquivo, que são usadas para marcar o início ou o fim da transferência de um arquivo. Essa função verifica se o pacote de controle é válido (seja de início ou de fim) e, se for válido, extrai as informações de tamanho do arquivo e nome do arquivo, armazenando-as nas variáveis apropriadas.

A função **readDataPacket** lê e analisa um pacote de dados armazenado no buffer de entrada *buf*. Ela extrai o número de sequência e os dados contidos no pacote de dados. Esta função verifica se o pacote de dados é válido e, caso seja válido, extrai o número de sequência e os dados, armazenando-os nas variáveis apropriadas.

A função **applicationLayer** é o ponto central do aplicativo e coordena a comunicação de dados. Ela inicializa o link layer, estabelece a conexão, define parâmetros como papel (TX ou RX), taxa de transmissão, número de tentativas e tempo limite. Dependendo do papel definido, esta função coordena a leitura ou escrita de pacotes de controle e dados. No transmissor (TX), ela lê um arquivo, cria pacotes de controle de início e fim, cria pacotes de dados e os envia pela camada de enlace. No receptor (RX), ela lê os pacotes recebidos, armazena os dados em um arquivo e espera até receber um pacote de controle de fim para encerrar a comunicação.

Validação

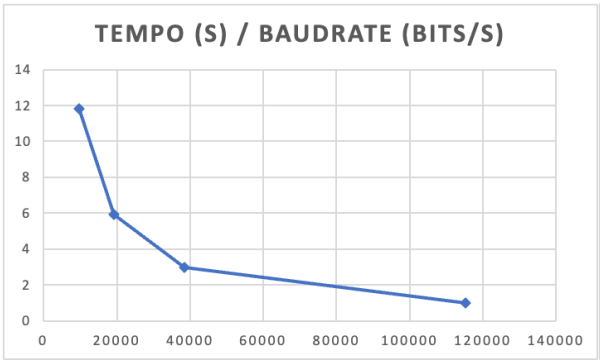
Testes foram realizados para testar o bom funcionamento do programa:

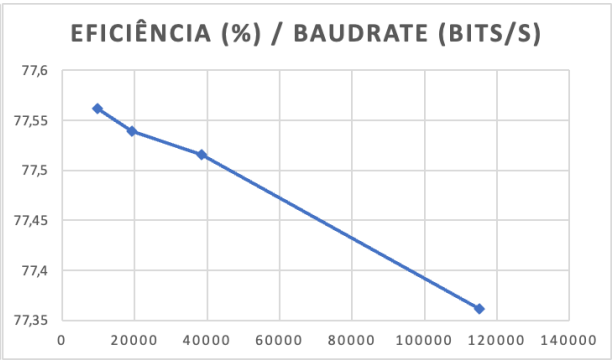
- Interrupção da porta série durante a execução
- Fazer ruído na porta série durante a execução
- Enviar ficheiros diferentes, tamanhos variados
- Enviar ficheiros com nomes diferentes
- Execução do programa com baudrate diferentes

Eficiência do protocolo de ligação de dados

Variação do Baudrate

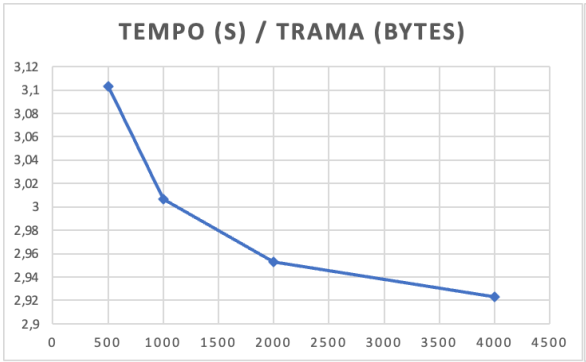
Com uma trama de tamanho fixo de 2000 bytes, testamos a variação do tempo e da eficiência em função do *baudrate*. Testamos com o baudrate a 9600, 19200, 38400 e 115200. Verificamos que a eficiência diminui com cada aumento do *baudrate*.

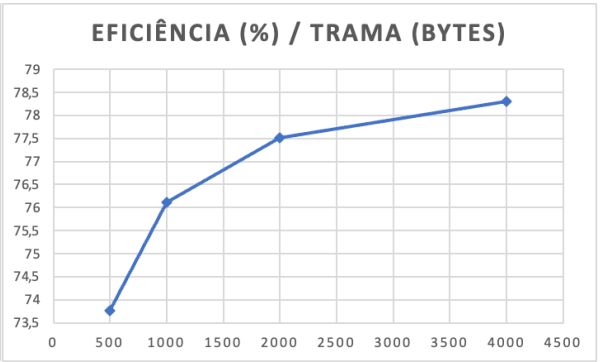




Variação do Tamanho da Trama

Com um *baudrate* fixo de 384000 bits/s, testamos a variação do tamanho da trama. Testámos com os valores: 4000, 2000, 1000 e 500. Verificamos que o tempo de transferência e o tamanho da trama são inversamente proporcionais. Notamos também que a eficiência aumenta com o tamanho da trama.





Conclusões

Ao longo do nosso relatório mostramos a solução que desenvolvemos face ao desafio proposto. Para demonstrarmos a nossa proposta de resolução mostramos as funções implementadas ao longo do código, os testes realizados e os respectivos resultados.

Conseguimos cumprir com sucesso o desafio proposto, tendo implementado com sucesso ambas as camadas : **LinkLayer** encarregou-se da interação com a porta série e de gerir os pacotes de informação e a **ApplicationLayer** que interagiu diretamente com o ficheiro a ser transferido.

No que toca a uma opinião mais pessoal relativamente ao projeto, podemos dizer que nos foi útil para consolidar a matéria lecionada nas aulas teóricas e desenvolver o nosso conhecimento em relação a conceitos como o byte stuffing, framing e o funcionamento do procedimento de Stop-and-Wait e como este detecta erros e lida com eles.

Anexo I - Código fonte

main.c

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

#include "application_layer.h"

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

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;
}
```

application_layer.h

```

#ifndef _APPLICATION_LAYER_H_
#define _APPLICATION_LAYER_H_

#include <stdio.h>

int createControlPacket(unsigned char* buf, int type, char* f, int
size);

int readControlPacket(unsigned char* buf, char* filename, int* filesz);

int createDataPacket(unsigned char* buf, int aux, int size, unsigned
char* data);

int readDataPacket(unsigned char* data, unsigned char* buf, int* seq);

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

#endif // _APPLICATION_LAYER_H_
```

application_layer.c

```
#include "application_layer.h"
#include "link_layer.h"
#include <string.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <termios.h>
#include <unistd.h>
#include <math.h>

extern int alarmFlag;
extern int alarmCounter;
extern int nr;
extern int valid;


int getFileSize(FILE* f)
{
   int size; // Variable to store the size of the file.

   // Seek to the end of the file to determine its size.
   fseek(f, 0, SEEK_END);
   size = (int)ftell(f);
   // Return to the beginning of the file.
   fseek(f, 0, SEEK_SET);
   return size; // Return the size of the file in bytes.
}

int getNumberBytes(int size)
{
   int counter = 0; // Initialize a counter to keep track of the number
of bytes.

   // Continuously divide the size by 256 until it becomes zero.
   while(size > 0) {
       size /= 256;
       counter++;
   }

   return counter; // Return the number of bytes required.
}
```

```
int createControlPacket(unsigned char* buf, int type, char* f, int
size) {
    if (type == 0) {
        buf[0] = C_START; // Start control packet type (0x02).
    } else if (type == 1) {
        buf[0] = C_END; // End control packet type (0x03).
    }

    unsigned L1 = getNumberBytes(size);
    unsigned L2 = strlen(f);
    unsigned tam = L1 + L2 + 5;

    // Byte 1: Reserved (always 0).
    buf[1] = 0;

    // Byte 2: L1 - Number of bytes required to represent the file size.
    buf[2] = L1;

    // Extract L1 bytes to represent the file size (big-endian).
    for (int i = 0; i < L1; i++) {
        int tmp = (size & 0x0000FFFF) >> 8;
        buf[3 + i] = tmp;
        size = size << 8;
    }

    // Byte L1+3: Reserved (always 1).
    int memo = L1 + 3;
    buf[memo] = 1;

    // Byte L1+4: L2 - Number of bytes required to represent the file
name.
    buf[memo + 1] = L2;

    // Copy the file name into the packet.
    memcpy(&buf[memo + 2], f, L2);

    // Log the bytes of the control packet for debugging.
    for (int i = 4; i < 4 + tam; i++) {
        fprintf(stderr, "0x%02X, ", buf[i]);
```

```
    }

    return tam;
}

int readControlPacket(unsigned char* buf, char* filename, int* filesz)
{
    *filesz = 0; // Initialize the file size.

    // Check if the packet is not a valid control packet (neither Start
nor End).
    if (buf[0] != 0x02 && buf[0] != 0x03) {
        fprintf(stderr, "Not a valid control packet.\n");
        return -1; // Return an error code.
    }

    // If it's a Start control packet (0x02).
    if (buf[1] == 0x00) {
        int fst = buf[2];
        for (int i = 0; i < fst; i++) {
            *filesz = *filesz * 256 + buf[3 + i]; // Extract and compute
the file size.
        }
    } else {
        return -1; // Return an error code for an invalid control
packet.
    }

    int snd = 0;
    int nxt = 5 + *filesz;

    // If it's a valid control packet with a filename.
    if (buf[nxt - 2] == 0x01) {
        snd = buf[nxt - 1];
        for (int j = 0; j < snd; j++) {
            filename[j] = buf[j + nxt]; // Extract the filename.
        }
    } else {
```

```
        return -1; // Return an error code for an invalid control
packet.
    }

    return 0; // Return success.
}

int createDataPacket(unsigned char* buf, int aux, int size, unsigned
char* data) {
    // Byte 0: Packet type (0x01 for data)
    buf[0] = DATA_PACKET;

    // Byte 1: Sequence number (limited to 0-255)
    buf[1] = aux % (BUF_SIZE - 1);

    // Byte 2-3: Data size
    buf[2] = size / BUF_SIZE;
    buf[3] = size % BUF_SIZE;

    //Printing the data content values
    for(int i = 4; i < 4 + size; i++) {

    }
    printf("Packet sent ");

    // Copy the data into the packet starting from Byte 4
    memcpy(buf + 4, data, size);

    // Calculate and return the total size of the constructed data
packet
    return 4 + size;
}

int readDataPacket(unsigned char* data, unsigned char* buf, int* seq)
{
    // Check if the packet is not a valid data packet.
    if (buf[0] != 0x01)
    {
        printf("Not a valid data packet.\n");
```

```
        return -1; // Return an error code.
    }

    *seq = buf[1]; // Extract and store the sequence number.
    int l2 = buf[2];
    int l1 = buf[3];
    int sz = (256 * l2) + l1; // Calculate the size of the data.

    for(int i = 0; i < sz; i++)
    {
        data[i] = buf[i + 4]; // Extract and store the data.
    }
    return sz; // Return the size of the extracted data.
}



void applicationLayer(const char *serialPort, const char *role, int
baudRate,
                      int nTries, int timeout, const char *filename)
{
    LinkLayer llObject;
    strcpy(llObject.serialPort, serialPort);

    // roles
    if (strcmp(role, "tx") == 0)
    {
        llObject.role = LlTx; // transmitter
    }
    else if (strcmp(role, "rx") == 0)
    {
        llObject.role = LlRx; // receiver
    }
    else
    {
        perror("Invalid role\n");
        exit(-1);
    }
```

```
    llObject.baudRate = baudRate;
    llObject.nRetransmissions = nTries;
    llObject.timeout = timeout;

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

    if (fd < 0) {
        perror("Connection error\n");
        exit(-1);
    }

    llopen(llObject); // Connection using the link layer struct.
    alarmReset(); //resets the alarm count and flag
    if (llObject.role == LlTx) {
        unsigned char control[MAX_PAYLOAD_SIZE];
        unsigned char buf[MAX_PAYLOAD_SIZE];

        // Open the file
        FILE* file = fopen(filename, "r");
        if (file == NULL) {
            perror("Error: file is not available.\n");
            exit(-1);
        }

        int size = getFileSize(file);

        int control_packet_size = createControlPacket(&control, 0,
filename, size);
        llwrite(fd, &control, control_packet_size);

        int bytes_r = 0;
        int sq = 0;
        alarmReset();
        while (bytes_r = fread(buf, 1, MAX_PAYLOAD_SIZE - 4, file)) {
            // Reading data from the file and creating data packets.
            unsigned char data[MAX_PAYLOAD_SIZE];
            int mount = createDataPacket(&data, sq, bytes_r, buf);
            llwrite(fd, &data, mount);
```

```
            sleep(3); //test with virtual cable to delay transmission
and allow to change cable status
            alarmReset();
            fprintf(stderr, "the size of the data packet sent is: %d\n",
mount);
            sq++;
        }

        fprintf(stderr, "Control Packet End sent:\n");
        control_packet_size = createControlPacket(&control, 1, filename,
size);
        llwrite(fd, &control, control_packet_size);
        alarmReset();
    }
    else if (llObject.role == LlRx) {
        int size;
        char sizec;
        unsigned char control[MAX_PAYLOAD_SIZE + 7];
        unsigned char data[MAX_PAYLOAD_SIZE + 7];

        int sz = llread(&control);
        printf("First llread");
        FILE* file2 = fopen(filename, "w");
        int ns = 0;

        while (1) {
            int sq;
            int lastnr = nr;

            do {
                sz = llread(&control);
            } while (sz == -1);

            fprintf(stderr, "New Packet created:\n");

            if (control[0] == 0x03) {
                fprintf(stderr, "Control Pack End received:\n");
                break;
            }
```

```
            else if (control[0] == 0x01) { //REJ0 frame : detects an
error by the receiver in the frame 0
                sz = readDataPacket(&data, &control, &sq);
                printf("\n\n");
                for (int i = 0; i < sz; i++) {
                    fprintf(stderr, "\\%02x", data[i]);
                }
                fwrite(data, 1, sz, file2);
            }
        }
    }
    llclose(0);
}
```

link_layer.h

```
#ifndef _LINK_LAYER_H_
#define _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>

#define BAUDRATE B38400
#define _POSIX_SOURCE 1

// SIZE of maximum acceptable payload.
// Maximum number of bytes that application layer should send to link
layer
#define MAX_PAYLOAD_SIZE 1000

#define BUF_SIZE 256
```

```
#define FALSE 0
#define TRUE 1

// MISC
#define FALSE 0
#define TRUE 1

//Data Packet
#define DATA_PACKET 0x01

//Delimitation constants (byte stuffing purpose)
#define FRAME_SIZE 5
#define A_SET 0x03
#define A_UA 0x01
#define FLAG 0x7E
#define ESC 0x7D
#define ESCE 0x5E
#define ESCD 0x5D
#define TR 0x03
#define REC 0x01
#define IFCTRL_ON 0x40
#define IFCTRL_OFF 0x00

//Control packet constants
#define C_START 2
#define C_END 3
#define C_FILE_SIZE 0
#define C_FILE_NAME 1
#define C_SET 0x03
#define C_DISC 0x0B
#define C_UA 0x07
#define C_REJ0 0x01 //rejected
#define C_REJ1 0x81
#define C_RR0 0x05 //received
#define C_RR1 0x85

typedef enum
{
   LlTx,
```

```
    LlRx,
} LinkLayerRole;

typedef enum
{
    START,
    FLAG_RCV,
    A_RCV,
    C_RCV,
    BCC1_RCV,
    STOP,
    C_INF,
    REJ
} LinkLayerState;

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

void stateMachine(LinkLayerState* status, unsigned char byte, int
type);

int llopen(LinkLayer connectionParameters);

void alarmController(int signal);

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

int sendReplyPacket();

int llread(unsigned char *packet);

int llclose(int showStatistics);
```

```
#endif // _LINK_LAYER_H_
```

link_layer.c

```
#include "link_layer.h"

// MISC
#define _POSIX_SOURCE 1 // POSIX compliant source
int alarmFlag = FALSE, alarmCounter = 0, timeout = 0,
retrans_data_counter = 0, ns = 0;
int check = TRUE, status_llwrite = 0, status_llread = 0, status_llclose
= 0, fd;
int valid = TRUE, nr = 1;
LinkLayer llObject;
struct termios oldtio, newtio;
unsigned char save[2] = {0};
LinkLayerState state = START; // Initial state

////////////////////////////////////////////////
// LLOPEN
////////////////////////////////////////////////

void stateMachine(LinkLayerState* status, unsigned char byte, int type)
{
   if(type == 0)
   {
       switch (*status)
       {
       case START:
           if(byte == FLAG) {
               *status = FLAG_RCV;
           }
           else {
               *status = START;
           }
           break;

       case FLAG_RCV:
```

```
            if(byte == TR || byte == REC) {

                save[0] = byte;
                *status = A_RCV;
            }
            else if(byte == FLAG) {
                *status = FLAG_RCV;
            }
            else {
                *status = START;
            }
            break;

        case A_RCV:
            if((byte == 0x07) || (byte == 0x03) || (byte == C_DISC) ||
(byte == 0x85) || (byte == 0x05)) {

                save[1] = byte;
                *status = C_RCV;
            }else if( (byte == 0x01) || (byte == 0x81) ){
                printf("BAD WRITE\n");
                save[1] = byte;
                *status = REJ;
            }
            else if(byte == FLAG) {
                *status = FLAG_RCV;
            }
            else if((byte == IFCTRL_ON) || (byte == IFCTRL_OFF)) {
                save[1] = byte;
                *status = C_INF;
            }
            else {
                *status = START;
            }
            break;

        case C_RCV:
            if(byte == (save[0] ^ save[1])) {
                printf("DONE XOR\n"); //se  A^C = BBC ?
```

```
                *status = BCC1_RCV;
                break;
            }
            else if(byte == FLAG) {
                *status = FLAG_RCV;
            }
            else {
                *status = START;
            }
            break;

        case BCC1_RCV:
            if(byte == FLAG) {
                *status = STOP;
            }
            else {
                *status = START;
            }
            break;

        case REJ:
            break;

        case STOP:
            break;

        default:
            break;
        }
    }
    else if(type == 1)
    {
        switch (*status)
        {
        case START:
            if(byte == FLAG) {
                *status = FLAG_RCV;
            }
            else {
```

```
            *status = START;
        }
        break;

    case FLAG_RCV:
        if(byte == TR || byte == REC) {
            save[0] = byte;
            *status = A_RCV;
        }
        else if(byte == FLAG) {
            *status = FLAG_RCV;
        }
        else {
            *status = START;
        }
        break;

    case A_RCV:
        if(byte == FLAG) {
            *status = FLAG_RCV;
        }
        else if((byte == IFCTRL_ON) || (byte == IFCTRL_OFF)) {
            save[1] = byte;
            *status = C_INF;
        }
        else {
            *status = START;
        }
        break;

    case C_INF:
        if(byte == (save[0]^save[1])) { //se  A^C = BBC ?
            *status = BCC1_RCV;
        }
        else if(byte == FLAG) {
            *status = FLAG_RCV;
        }
        else {
            *status = START;
```

```
            }
            break;

        case BCC1_RCV:
            if(byte == FLAG) {
                *status = STOP;
            }
            break;

        case STOP:
            break;

        default:
            break;
        }
    }
}

int sendSet()
{
    unsigned char buf[BUF_SIZE + 1] = {0};
    buf[0] = FLAG;
    buf[4] = FLAG;
    buf[1] = TR;
    buf[2] = C_SET;
    buf[3] = buf[1] ^ buf[2];
    (void)signal(SIGALRM, alarmController);

    while (state != STOP && alarmCounter < (llObject.nRetransmissions +
1))
    {
        if (!alarmFlag)
        {
            printf("Set Written\n");
            state = START;
            write(fd, buf, 5);
            alarm(llObject.timeout);
            alarmFlag = TRUE;
        }
```

```
        if (read(fd, buf, 1) > 0) {
            stateMachine(&state,buf[0],0);
            if(state == STOP)
            {
                printf("Read UA\n");
                break;
            }
        }
    }
    return alarmCounter;
}

int sendUAreadSet()
{

unsigned char buffer[BUF_SIZE +1] = {0};
    while(TRUE)
    {
        int bytes = read(fd,buffer,1);
        stateMachine(&state, buffer[0], 0);
        if(state == STOP)
        {
            printf("Finished Reading SET\n");
            break;
        }
    }
    buffer[0] = FLAG;
    buffer[1] = A_UA;
    buffer[2] = C_UA;
    buffer[3] = buffer[1] ^ buffer[2];
    buffer[4] = FLAG;
    int bytes = write(fd,buffer,5);
    return 1;
}

void alarmController(int signal) {
    alarmFlag = FALSE;    // alarm flag -> TRUE when the alarm has
occured
```

```
    alarmCounter++;
    printf(" alarme\n");
}

int alarmReset() {
    alarmCounter = 0;
    alarmFlag = FALSE;

}

int llopen(LinkLayer connectionParameters)
{
    llObject = connectionParameters;
    fd = open(llObject.serialPort, O_RDWR | O_NOCTTY);
    if (fd < 0)
    {
        perror(llObject.serialPort);
        exit(-1);
    }

    // Save current port settings
    if (tcgetattr(fd, &oldtio) == -1)
    {
        perror("tcgetattr");
        exit(-1);
    }

    // Clear struct for new port settings
    memset(&newtio, 0, sizeof(newtio));

    newtio.c_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;
    newtio.c_iflag = IGNPAR;
    newtio.c_oflag = 0;

    // Set input mode (non-canonical, no echo)
    newtio.c_lflag = 0;
    newtio.c_cc[VTIME] = 0; // Inter-character timer unused
    newtio.c_cc[VMIN] = 0.1;  // Blocking read until 5 chars received
```

```
    //   TCIFLUSH - flushes data received but not read.
    tcflush(fd, TCIOFLUSH);

    // Set new port settings
    if (tcsetattr(fd, TCSANOW, &newtio) == -1)
    {
        perror("tcsetattr");
        return -1;
    }

    unsigned char byte;
    timeout = llObject.timeout;
    retrans_data_counter = llObject.nRetransmissions;
    switch(llObject.role) {
        case LlTx:{
            sendSet();
            break;
        }
        case LlRx:{
            sendUAreadSet();
            break;
        }
        default:
            return -1;
            break;
    }
return 1;
}

////////////////////////////////////////////////
// LLWRITE
////////////////////////////////////////////////
int llwrite(int fd, const unsigned char *buf, int bufSize)
{
  unsigned char trama_info[2*MAX_PAYLOAD_SIZE];

    trama_info[0] = FLAG;
    trama_info[1] = A_SET;
    trama_info[2] = (ns == 0) ? 0x00 : 0x40;
```

```
trama_info[3] = trama_info[1] ^ trama_info[2];

int trama_info_size = 0;
int bcc_2 = 0;

bcc_2 ^= buf[0];
bcc_2 ^= buf[1];
bcc_2 ^= buf[2];
bcc_2 ^= buf[3];

for(int j = 4; j < bufSize; j++)
{
    bcc_2 ^= buf[j];
}

for(int i = 0; i < bufSize; i++) {
    if(buf[i] == FLAG) {
        trama_info[4+trama_info_size] = ESC;
        trama_info[5+trama_info_size] = ESCE;
        trama_info_size+= 2;
    }
    else if(buf[i] == ESC) {
        trama_info[4+trama_info_size] = ESC;
        trama_info[5+trama_info_size] = ESCD;
        trama_info_size+= 2;
    }
    else {
        trama_info[4+trama_info_size] = buf[i];
        trama_info_size++;
    }
}
if(bcc_2 == FLAG) {
    trama_info[4+trama_info_size] = ESC;
    trama_info[5+trama_info_size] = ESCE;
    trama_info[6+trama_info_size] = FLAG;
    trama_info_size+= 1;
}
else if(bcc_2 == ESC) {
    trama_info[4+trama_info_size] = ESC;
```

```
        trama_info[5+trama_info_size] = ESCD;
        trama_info[6+trama_info_size] = FLAG;
        trama_info_size+= 1;
    }
    else {
        trama_info[4+trama_info_size] = bcc_2;
        trama_info[5+trama_info_size] = FLAG;
    }

    trama_info_size += 6;

    unsigned char buf_aux[BUF_SIZE+1] = {0};

    (void)signal(SIGALRM,alarmController);
    while(status_llwrite != STOP && alarmCounter <
(llObject.nRetransmissions + 1))
    {
        //printf("The value of the alarmFlag %d\n", alarmFlag);
        //printf("status_llwrite == REJ%d\n",status_llwrite == REJ);
definition of the link-connection
        //printf("the value of the llObject.timeout %d\n",
llObject.timeout);
        if(!alarmFlag || status_llwrite == REJ) //|| status_llwrite ==
REJ)
        {
            status_llwrite = START;
            fprintf(stderr,"Sent Packet: %d\n",ns);
            int byte_written = write(fd,trama_info,trama_info_size);
            alarm(llObject.timeout);
            alarmFlag = TRUE;
        }
        int k = 0;
        unsigned char temp;
    if(read(fd,buf_aux + k,1) > 0)
    {
        fprintf(stderr,"Reading state: %d", status_llwrite);

        stateMachine(&status_llwrite, buf_aux[k], 0);
        if(status_llwrite == REJ)
```

```
        {
            printf("REJ state reached");
            read(fd,buf_aux + k,1);
            k++;
            read(fd,buf_aux + k,1);
            k++;
        }
        fprintf(stderr,"Buf: %02x \n", buf_aux[k]);
        k++;
        printf("status: %d\n", status_llwrite);
        if(status_llwrite == STOP)
        {
            printf("STOP reached\n");
            ns = (1+ns) % 2;
            valid = TRUE;
            break;
        }
    }
    }
    status_llwrite = START;
    alarmFlag = FALSE;
    if(alarmCounter >= (llObject.nRetransmissions + 1))
    {
        alarmCounter = 0;
        llclose(0);
        exit(1);
    }
    return trama_info_size;
}

////////////////////////////////////////////////
// LLREAD
////////////////////////////////////////////////

int sendReplyPacket() {
    unsigned char out[6];

    // Set the beginning and ending flag bytes.
    out[0] = FLAG;
```

```
    out[4] = FLAG;

    // Define the packet type (command).
    out[1] = TR;

    // Determine the REJ type based on the frame number (nr).
    if (nr) out[2] = C_REJ1; // If nr = 1, use C_REJ1.
    else out[2] = C_REJ0; // If nr = 0, use C_REJ0.

    // Calculate and set the BCC (Block Check Character) field.
    out[3] = out[1] ^ out[2];

    // Send the constructed REJ packet to the serial port.
    write(fd, out, 5);

    return 5; // The size of the sent REJ packet is always 5 bytes.
}

int llread(unsigned char *packet) {
    unsigned char buf[MAX_PAYLOAD_SIZE * 2]; // Temporary buffer for
received data.
    unsigned char initialPack[MAX_PAYLOAD_SIZE]; // Buffer for storing
the initial packet data.

    int sz = 0;
    status_llread = START;
    memset(packet, 0, sizeof(packet)); // Initialize the packet buffer.

    while (1) {
        if (read(fd, buf + sz, 1) > 0) {
            if (sz + 1 > MAX_PAYLOAD_SIZE * 2) {
                // If the flag couldn't be found in the end packet, send
a REJ packet.
                printf("Flag not found at the end of the packet\n");
                sendReplyPacket();
                return -1;
            }
            stateMachine(&status_llread, buf[sz], 1); // Process the
received data.
```

```
            sz++;
            if (status_llread == STOP) {
                printf("Received complete frame\n");
                break;
            }
        }
    }
    status_llread = START;

    int bcc_2 = 0;
    if (buf[sz - 3] == ESC && buf[sz - 2] == ESCD) {
        bcc_2 = ESC;
        sz--;
    } else if (buf[sz - 3] == ESC && buf[sz - 2] == ESCE) {
        bcc_2 = FLAG;
        sz--;
    } else {
        bcc_2 = buf[sz - 2];
    }

    for (int l = 0; l < sz - 6; l++) {
        initialPack[l] = buf[4 + l];
    }

    // Extract data from the received frame and handle escape sequences.
    packet[0] = initialPack[0];
    packet[1] = initialPack[1];
    packet[2] = initialPack[2];
    packet[3] = initialPack[3];

    int j = 4;
    for (int i = 4; i < sz - 6; i++) {
        if (initialPack[i] == ESC && initialPack[i + 1] == ESCE) {
            packet[j++] = FLAG;
            i++;
        } else if (initialPack[i] == ESC && initialPack[i + 1] == ESCD)
{
            packet[j++] = ESC;
            i++;
```

```
        } else packet[j++] = initialPack[i];
    }

    int bcc_cal = 0;
    bcc_cal ^= packet[0];
    bcc_cal ^= packet[1];
    bcc_cal ^= packet[2];
    bcc_cal ^= packet[3];

    for (int i = 4; i < j; i++) {
        bcc_cal ^= packet[i];
    }

    unsigned char outbuf[6]; // Buffer for constructing response frame.
    outbuf[0] = FLAG;
    outbuf[1] = TR;
    outbuf[4] = FLAG;

    if(bcc_cal == bcc_2){
        nr = (nr + 1) % 2;
        if(nr) {
            outbuf[2] = C_RR1;
        }
        else {
            outbuf[2] = C_RR0;
        }
    }else if(bcc_cal != bcc_2){
        if(nr) {
            outbuf[2] = C_REJ1;
        }
        else {
            outbuf[2] = C_REJ0;
        }
        printf("Bad packet detected\n");
        outbuf[3] = outbuf[1] ^ outbuf[2];
        write(fd, outbuf, 5);
        return -1;
    }
```

```
    outbuf[3] = outbuf[1] ^ outbuf[2];
    int bytes = write(fd, outbuf, 5); // Send the response frame.

    printf("Response frame sent\n");

    int sz_final = j;
    return sz_final; // Return the size of the received packet.
}

sendDiscCommand()
{
    unsigned char buf[BUF_SIZE + 1] = {0};
    buf[0] = FLAG;
    buf[1] = TR;
    buf[2] = C_DISC;
    buf[3] = buf[1] ^ buf[2];
    buf[4] = FLAG;
    (void)signal(SIGALRM,alarmController);
    while(status_llclose != STOP && alarmCounter <
(llObject.nRetransmissions + 1))
    {
        if(!alarmFlag)
        {
            printf("Disc Written\n");
            status_llclose = START;
            write(fd,buf,5);
            alarm(llObject.timeout);
            alarmFlag = TRUE;
        }
        if(read(fd,buf,1) > 0 )
        {
            stateMachine(&status_llclose, buf[0], 0);
            if(status_llclose == STOP)
            {
                printf("READ  Disc\n");
                break;
            }
        }
    }
```

```
    return alarmCounter;
}

////////////////////////////////////////////////
// LLCLOSE
////////////////////////////////////////////////
int llclose(int showStatistics) {
    alarmFlag = FALSE;
    alarmCounter = 0;

    switch (llObject.role) {
        case LlTx:
            if (sendDiscCommand() == 3) {
                return -1; // Error sending DISC command.
            }

            // Send UA (Unnumbered Acknowledgment).
            unsigned char buf[BUF_SIZE + 1];
            buf[0] = FLAG;
            buf[1] = REC;
            buf[2] = C_UA;
            buf[3] = buf[1] ^ buf[2];
            buf[4] = FLAG;
            write(fd, buf, 5);

            printf("Sent UA\n");
            break;

        case LlRx:
        if(1 == 2){}
            // Receive DISC command.
            unsigned char buf2[BUF_SIZE + 1];
            while (1) {
                int bytes = read(fd, buf2, 1);
                stateMachine(&status_llclose, buf2[0], 0);
                if (status_llclose == STOP) {
                    printf("READ DISC COMPLETED\n");
                    break;
                }
```

```
            }
            // Send DISC response.
            buf2[0] = FLAG;
            buf2[1] = REC;
            buf2[2] = C_DISC;
            buf2[3] = buf2[1] ^ buf2[2];
            buf2[4] = FLAG;
            int bytes = write(fd, buf2, 5);
            printf("Sent DISC\n");

            unsigned char buf3[BUF_SIZE + 1];
            unsigned int counter = 0;
            while (1) {
                int bytes = read(fd, buf3, 1);
                stateMachine(&status_llclose, buf3[0], 0);
                if (status_llclose == STOP) {
                    printf("READ UA COMPLETED\n");
                    break;
                }
                counter++;
            }
            break;
        default:
            break;
    }

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

    // Close the serial port.
    close(fd);
    return 1; // Successful closure.
}
```