Protocolo de Ligação de Dados

Relatório do 1º Trabalho Laboratorial

Redes de Computadores – FEUP

Duarte Alexandre Pinto Brandão (ei10060@fe.up.pt)

João Miguel Dias Ferreira Gouveia (ei12063@fe.up.pt)

Ricardo Oliveira Neto Leite (up200902919@fe.up.pt)

Sumário

Este relatório foi criado com base no primeiro trabalho prático da cadeira de Redes de Computadores, do MIEIC.

Os autores esperam que seja útil e que satisfaça os requisitos do(s) professor(es) que o va(i/õ) avaliar.

Introdução

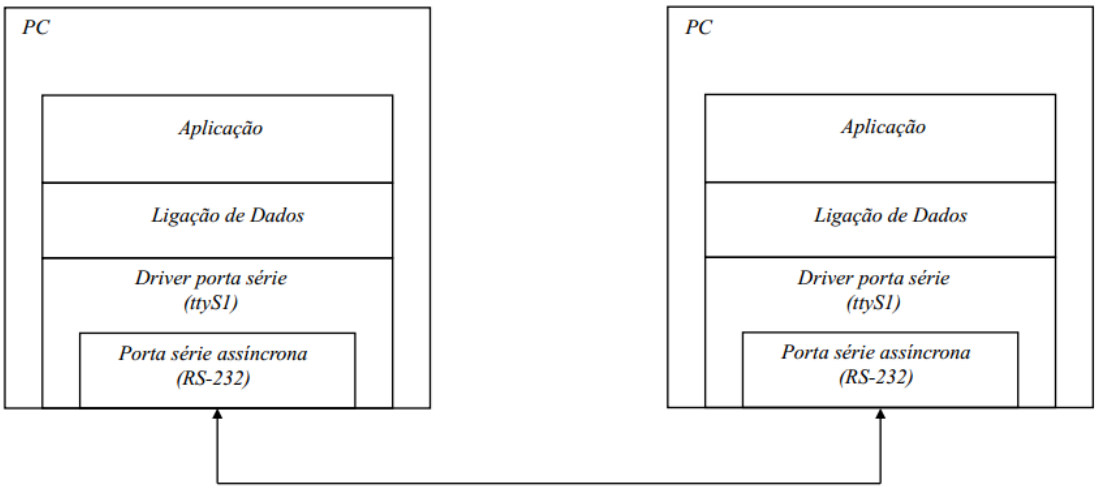
O principal objetivo deste trabalho é a implementação, e subsequente teste, de um protocolo de ligação de dados, utilizando métodos pré-definidos. Desse modo, dois sistemas (computadores ou máquinas virtuais) poderão executá-lo ao mesmo tempo de uma forma cooperativa, sendo um o emissor de dados e o outro o recetor.

No que toca ao relatório, este tem a função de apresentar uma descrição detalhada deste trabalho, seguindo uma estrutura específica que consiste, nesta ordem, de uma introdução, de um desenvolvimento (formado por diversas características do projeto, como a estrutura do código e os casos de uso principais) e de uma conclusão. Deste modo, pode servir como um manual de instruções ao projeto em si.

Arquitetura

Para o projeto, foram necessárias duas instâncias do sistema operativo Linux, com especial concentração na sua linha de comandos (ou "Terminal"). Também foi indispensável o uso de um driver para a porta de série (ttyS0/ttyS1) para o sistema de ligação/comunicação de dados ser executado com sucesso.

Para o código em si, foi utilizada a linguagem de programação C.



Estrutura do código

O código do trabalho tem a seguinte estrutura:

- Makefile - Um comando para inicializar o programa.
- main.c – O ficheiro-classe base do código, tem o papel de agir como núcleo e chamar as funções presentes noutros ficheiros do código. (função principal: startApp())
- definitions.h – O ficheiro header que define termos em certos números e operações. A sua utilidade é tornar o código mais simples.
- application.c – O ficheiro-classe dedicado ao protocolo de aplicação. (funções principais: setup(), sendFile(), receiveFile(), sendControl(), receiveControl(), sendPacket(), cliPacket(), cliBaudrate(), cliRetries(), cliTimeout())
- application.h – O ficheiro header dedicado ao protocolo de aplicação.
- link.c – O ficheiro-classe dedicado ao protocolo de ligação lógica. Possui o código que vai ser transferido de um sistema para outro. (funções principais: llopen(), llwrite(), llread(), llclose())
- link.h – O ficheiro header dedicado ao protocolo de ligação lógica.

Casos de uso principais

Perspectiva geral:

1. Seleccionar papel (emissor, receptor, saír)

Perspectiva do emissor:

1. Seleccionar status correspondente ao emissor
2. Introduzir a porta de série (valor entre 0 e 4)
3. Seleccionar baudrate (listagem com 7 opções)
4. Introduzir o timeout (valor maior do que zero)
5. Introduzir o número de tentativas (valor maior do que zero)
6. Introduzir o caminho do ficheiro a enviar
7. Introduzir o tamanho dos pacotes de dados

Perspectiva do receptor:

1. Seleccionar status correspondente ao receptor
2. Introduzir a porta de série (valor entre 0 e 4)
3. Seleccionar baudrate (listagem com 7 opções)

Sequências de chamada de funções:

A aplicação é inicializada e chama a função setup() que por sua vez vai chamar a função cliStatus() que corresponde ao caso de utilização 1. O programa vai requisitar ao utilizador

para escolher três opções (emissor, receptor, saír). Caso o utilizador não seleccione a última opção o programa chama a função cliPort() que corresponde ao caso de utilização 2, caso contrário o programa termina. O programa avança e chama a função cliBaudrate() que corresponde ao caso de utilização 3. Na perspectiva do receptor, o programa chama a função receiveFile() após seleccionar o baudrate, enquanto que, na perspectiva do emissor, o programa chama a função cliTimeout() e depois cliRetries() que correspondem aos casos de utilização 4 e 5. No caso de utilização 6 o programa chama a função cliChooseFile() e no caso 7 chama a função cliPacket(). Por fim é chamada a função sendFile() que vai proceder ao envio do ficheiro.

Protocolo de Ligação Lógica

Fase de Estabelecimento

- Inicializar a struct linkLayer
- Iniciar a ligação com a porta de série
- Envio/Recepção de Tramas SET
- Envio/Recepção de Tramas UA

Fase de Transferência de Dados

- Envio de Tramas I (função **send_data()**) na qual o campo de controlo bcc2 é gerado (função **generate_bcc2()**) e é utilizado o mecanismo de byte stuffing (função **byteStuffing()**)
- Envio de Tramas RR (função **send_rr()**)
- Envio de Tramas REJ (função **send_rej()**)
- Processamento da resposta após enviar uma Trama I através da função **rec_resp_receiver()** (recepção de tramas REJ, RR). Caso ocorra timeout faz a retransmissão da Trama I
- Recepção de Tramas I (através da função **rec_data()**) na qual o campo de controlo bcc2 é verificado (utilização da função **byteDestuffing()**) e a função procede a uma resposta adequada (envia RR ou REJ).
- Configuração do Temporizador
- Confirmação/Controlo de Erros através de Stop-and-Wait

Fase de Terminação

- Envio/Recepção de Tramas DISC
- Envio/Recepção de Tramas UA
- Fechar a ligação com a porta de série

Protocolo de Aplicação

- **Selecção de parametros por parte do utilizador**
  Esta funcionalidade foi conseguida através da implementação de várias funções que lêem os valores introduzidos pelo utilizador e por sua vez estes valores são definidos nas estrututas da aplicação (struct app_layer) e da ligação de dados (struct linkLayer).
- **Envio/Recepção do ficheiro**
  - Envio/Recepção de pacotes de controlo
  - Envio/Recepção de pacotes de dados
  - Criar ficheiro a receber
  - Escrever no ficheiro a receber

  Para o envio do ficheiro implementou-se a função **sendFile()** que chama a função **sendControl()** de forma a enviar os pacotes de controlo necessários à sinalização do ínicio e fim da transmissão. Para a transmissão dos pacotes de dados do ficheiro implementou-se a função **sendPacket()** que é chamada pela função **sendFile()**.

  Para a recepção do ficheiro implementou-se a função **receiveFile()** que chama a função **receiveControl()** de forma validar a recepção do pacote de controlo que sinaliza o início da transmissão do ficheiro. Implementou-se a função **createFile()** que é chamada pela função **receiveFile()** para criar o ficheiro a receber. Após a criação do ficheiro é chamada a função **processPacket()** que recebe cada pacote de dados e escreve para o ficheiro a informação através da função **writeToFile**().

Validação

Foram efectuados três testes diferentes:

- **Teste Normal:** Correu-se a aplicação em condições normais, onde não foram encontradas quaisquer problemas ao enviar/receber o ficheiro.
- **Teste retirando o cabo de ligação:** Retirou-se o cabo de ligação na fase de transmissão dos pacotes de dados, na qual a aplicação activou o alarme e iniciou o processo de tentativas de retransmissão dos mesmos. Voltou-se a ligar o cabo e aplicação enviou com sucesso o ficheiro.
- **Teste colocando uma chave no modem:** Obteve-se o mesmo resultado ao do teste do cabo de ligação.

Elementos de valorização

- Selecção de parâmetros pelo utilizador (baudrate, packet size, tries, timeout)
- Implementação de REJ
- Verificação do tamanho e do nome do ficheiro recebido
- Verificação de pacotes perdidos ou duplicados

Conclusões

O objectivo deste trabalho era implementar um protocolo de ligação de dados que consistia em fornecer um serviço de transmissão de dados entre dois computadores ligados através de um cabo série e testar o protocolo com uma aplicação simples de transferência de ficheiros.

Para tal, ao implementar o protocolo de ligação de dados foi necessário criar funções para o envio e recepção de diferentes tipos de tramas, de forma a que fosse possível estabelecer/terminar a ligação e fazer a transmissão/recepção da informação. Para que a comunicação da informação fosse fiável implementou-se mecanismos de controlo de erros e controlo de fluxo.

Na parte da aplicação foi necessário implementar toda a interatividade entre o utilizador e criar funções para aceder ao serviço da ligação de dados.

Este trabalho foi bem sucedido, uma vez que com a implementação dos diferentes blocos funcionais e da interface apresentados foi possível criar uma aplicação de teste que envia um ficheiro de um computador para o outro através da porta de série.

ANEXO I

application.c

```
#include "application.h"


app_layer_T settings;

int corrupted_data = 0;


int startApp() {
	start_settings();
	setup();
}


int setup() {
	//need to setup status(sender/receiver), port, timeout and retries


	settings.status = cliStatus();


	while (settings.status != EXIT) {
		settings.port = cliPort();
		setBR(cliBaudrate());
		if (settings.status == SENDER) {
			setTO(cliTimeout());
			setNT(cliRetries());
			//TODO:check this var
			unsigned char *path = cliChooseFile();
			settings.fileSize = getFileSize(path);


			settings.packetSize = cliPacket();
			settings.fileDescriptor = open(path, O_RDONLY);
			sendFile();
			free(path);
		} else {
```

```
                        receiveFile();
                }
        }
        return 0;
}

int receiveFile() {
        printf("Waiting for connection!\n");
        settings.serialPortDescriptor = llopen(settings.port, settings.status);
        if (settings.serialPortDescriptor == -1) {
                printf("error opening connection! \n");
                return -1;
        }

        receiveControl(2);

        printf("Name: %s\n",settings.name);
        if ((settings.fileDescriptor = createFile(settings.name)) == -1) {
                printf("error creating file!\n");
                return -1;
        }

        int value = 0;
        int last_size = 0;
        unsigned char packet[PACKET_MAX_SIZE];

        while (TRUE) {
                value = llread(settings.serialPortDescriptor, packet);
                if (value == DISCONECTED) {
                        if (verifyFile(packet, last_size)) {
                                printf("Error receiving packets!\n");
                        }
                        break;
```

```
                } else {
                        last_size = value;
                        processPacket(packet);
                }
        }

        close(settings.fileDescriptor);

        if (corrupted_data || value == 1) {
                printf("Data corrupted, discarding!\n");
        }

        llclose(settings.serialPortDescriptor, settings.status);
        return 0;

}

int verifyFile(unsigned char * packet, unsigned int size) {
        unsigned int pos = 0;
        struct stat stat_file;

        unsigned int ctrlField = packet[pos++];

        if (ctrlField != 3) {
                return 1;
        }

        unsigned char fileName[MAX_STRING_SIZE] = { 0 };
        unsigned int fileSize = 0;

        while (pos < size) {
                switch (packet[pos]) {
                case 0:
```

```
                pos++;
                memcpy(&fileSize, &packet[pos + 1], packet[pos]);
                pos += packet[pos] + 1;

                if (settings.fileSize != fileSize) {
                        return 1;
                }

                if (fstat(settings.fileDescriptor, &stat_file) == -1) {
                        return 1;
                }

                if (stat_file.st_size != fileSize) {
                        return 1;
                }

                break;

        case 1:

                pos++;
                memcpy(&fileName, &packet[pos + 1], packet[pos]);
                pos += packet[pos] + 1;

                if (strcmp((char*) fileName, (char *) settings.name)
                                != 0) {
                        return 1;
                }
                break;
        }
    }
    return 0;
}
```

```
int processPacket(unsigned char * packet) {
	unsigned int pos = 0;
	unsigned int number = 0;
	unsigned int temp = 0;
	unsigned int size = 0;
	unsigned char * info;

	unsigned int ctrlField = packet[pos++];
	if (ctrlField == 1) {

		number = packet[pos++];
		temp = number - settings.currentNum;
		if (temp == -254) {
			temp = 1;
		}
		if (temp == 1) {
			size = packet[pos++]*256;
			size += packet[pos++];

			info = malloc(size + 1);
			memset(info, 0, size + 1);
			memcpy(info, &packet[pos], size);

			if (writeToFile(info, size) == -1) {
				printf("Error writing to file\n");
				return -1;
			}
			settings.currentNum = number;

			free(info);
		}
	}/* else {
```

```
                corrupted_data = -1;
        }*/

        return 0;
}

int writeToFile(unsigned char * info, unsigned int byn_number) {
        if (write(settings.fileDescriptor, info, byn_number)
                        != byn_number) {
                printf("error in write! \n");
                return -1;
        }
        return byn_number;
}

int createFile(char * name) {
        int fd = open((char *) name, O_CREAT | O_EXCL | O_WRONLY,
                        S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH);

        return fd;
}

int receiveControl(int var) {
        unsigned char packet[PACKET_MAX_SIZE] = { 0 };
        int size = 0;
        int pos = 0;

        size = llread(settings.serialPortDescriptor, packet);
        if (size == -1) {
                return -1;
        }

        if (packet[pos] != var) {
```

```
                printf("Error in receive control!\n");
                return -1;
        }

        pos++;

        while (pos < size) {
                switch (packet[pos]) {
                case 0:
                        pos++;
                        memcpy(&settings.fileSize, &packet[pos + 1], packet[pos]
                        );
                        pos += sizeof(unsigned int) + 1;
                        printf("Filesize %d\n",settings.fileSize);
                        break;
                case 1:
                        pos++;
                        memcpy(&settings.name, &packet[pos + 1],
                                        packet[pos]);
                        pos += strlen((char *) settings.name) + 2;

                        printf("Filename %s\n",settings.name);
                        break;
                }
        }
        return 0;
}

int sendFile() {
        settings.serialPortDescriptor = llopen(settings.port,
                        settings.status);
        if (settings.serialPortDescriptor == -1) {
                perror("error opening link!!\n\n");
```

```
                close(settings.fileDescriptor);
                return -1;
}

if (sendControl(2) == -1) {
                printf("Error in control packet\n");
                close(settings.fileDescriptor);
                return -1;
}
printf("control returned!\n");
int stop = 0;
unsigned char* packet = malloc(settings.packetSize);
unsigned int size = 0;

while (!stop) {
                printf("read!\n");
                size = read(settings.fileDescriptor, packet,
                                settings.packetSize);
                if (size != settings.packetSize) {
                                stop = -1;
                }
printf("PACKET SIZE :%d\n",size);
                if (sendPacket(packet, size) == -1) {
                                printf("Error transmiting packet \n");
                                close(settings.fileDescriptor);
                                return -1;
                }
}

if (sendControl(3) == -1) {
                printf("Error closing link\n");
                close(settings.fileDescriptor);
                return -1;
```

```
        }

        llclose(settings.serialPortDescriptor, settings.status);
        return 0;
}

int sendPacket(unsigned char * message, unsigned int size) {
        unsigned char packet[PACKET_MAX_SIZE];
        unsigned int pos = 0;

        packet[pos++] = 1;
        settings.currentNum = (settings.currentNum % 255) + 1;
        packet[pos++] = settings.currentNum;

        packet[pos++] = size/256;
        packet[pos++] = size%256;
        memcpy(&packet[pos], message, size);
        return llwrite(settings.serialPortDescriptor, packet,
                        (size + 4));
}

int sendControl(int var) {
        unsigned char packet[PACKET_MAX_SIZE] = { 0 };
        unsigned int size = 0;
        packet[0] = var;
        size++;

        uint8_t fileSize = 0;
        packet[size++] = fileSize;
        uint8_t length = sizeof(unsigned int);
        packet[size++] = length;
        memcpy(&packet[size], &settings.fileSize,
                        sizeof(unsigned int));
```

```
        size += sizeof(unsigned int);


        uint8_t fileName = 1;
        packet[size++] = fileName;
        length = strlen((char *) settings.name) + 1;
        packet[size++] = length;
        memcpy(&packet[size], &settings.name,
                        strlen((char *) settings.name) + 1);
        size += strlen((char *) settings.name) + 1;
        printf("filename: %s\n",settings.name);
        return llwrite(settings.serialPortDescriptor, packet, size);

}

int cliBaudrate() {

        unsigned int choice = 0;
        int baud[] = { B110, B300, B600, B1200, B2400, B4800, B9600};
        printf("Select the baudrate: \n");
        printf("1) B110 \n");
        printf("2) B300 \n");
        printf("3) B600 \n");
        printf("4) B1200 \n");
        printf("5) B2400 \n");
        printf("6) B4800 \n");
        printf("7) B9600 \n");

        char temp[MAX_STRING_SIZE];
        fgets((char *) temp, MAX_STRING_SIZE, stdin);
        sscanf(temp, "%d", &choice);

        while (choice < 1 || choice > 7) {
                printf("Incorrect option\n");
```

```
                fgets((char *) temp, MAX_STRING_SIZE, stdin);
                sscanf(temp, "%d", &choice);
        }
        return baud[choice - 1];
}

int cliPacket() {
        printf("Enter the size of the data packets: \n\n");
        int choice = -1;
        char temp[MAX_STRING_SIZE];

        fgets((char*) temp, MAX_STRING_SIZE, stdin);
        sscanf(temp, "%d", &choice);

        while (choice < 1 || choice > PACKET_MAX_SIZE
                        || choice > (settings.fileSize) * pow(2, 8)) {
                printf("Invalid Size!\n\n");
                fgets((char*) temp, MAX_STRING_SIZE, stdin);
                sscanf(temp, "%d", &choice);
        }

        return choice;

}

int cliRetries() {
        printf("Enter the number of tries: \n\n");
        int choice = -1;
        char temp[MAX_STRING_SIZE];

        fgets((char*) temp, MAX_STRING_SIZE, stdin);
        sscanf(temp, "%d", &choice);
```

```
	while (choice < 0) {
		printf("Tries must be positive!\n\n");
		fgets((char*) temp, MAX_STRING_SIZE, stdin);
		sscanf(temp, "%d", &choice);
	}

	return choice;
}

char* cliChooseFile() {
	printf("What is the path for the file?\n\n");

	unsigned char *path = malloc(PATH_MAX);
	memset(path, 0, PATH_MAX);
	fgets((char *) path, PATH_MAX, stdin);
	path[strlen((char *) path) - 1] = '\0';
	printf("%s\n", path);
	while (validPath(path)) {
		memset(path, 0, PATH_MAX);
		fgets((char *) path, PATH_MAX, stdin);
		path[strlen((char *) path) - 1] = '\0';
		printf("Invalid path!");
	}
struct stat file_stat;
	stat((char ) path, &file_stat);
		strcpy((char *) settings.name, basename((char *) path));
	printf("path is valid\n");
	printf("filename %s", settings.name);
	return path;
}

int getFileSize(char* path) {
	FILE* file = fopen(path, "rb");
```

```
	fseek(file, 0L, SEEK_END);
	int sz = ftell(file);

	fseek(file, 0L, SEEK_SET);

	return sz;
}

int validPath(unsigned char * path) {
	FILE* file = fopen(path, "rb");

	return !file;
}

int cliTimeout() {
	printf("Enter a timeout value to use(seconds): \n\n");
	int choice = -1;
	char temp[MAX_STRING_SIZE];

	fgets((char*) temp, MAX_STRING_SIZE, stdin);
	sscanf(temp, "%d", &choice);

	while (choice < 0) {
		printf("Time must be positive!\n\n");
		fgets((char*) temp, MAX_STRING_SIZE, stdin);
		sscanf(temp, "%d", &choice);
	}

	return choice;
}

int cliPort() {
```

```
		printf("Enter a Serial Port to use: \n\n");


		int choice = -1;

		char temp[MAX_STRING_SIZE];


		fgets((char*) temp, MAX_STRING_SIZE, stdin);

		sscanf(temp, "%d", &choice);


		while (choice < 0 || choice > 4) {

			printf("Invalid Port!\n\n");

			fgets((char*) temp, MAX_STRING_SIZE, stdin);

			sscanf(temp, "%d", &choice);

		}


		return choice;

}


int cliStatus() {

		printf("\n Select a role: \n");

		printf("1) Sender \n");

		printf("2) Receiver \n");

		printf("3) EXIT \n\n");


		int choice = -1;


		char temp[MAX_STRING_SIZE];


		fgets((char*) temp, MAX_STRING_SIZE, stdin);

		sscanf(temp, "%d", &choice);


		while (choice != SENDER && choice != RECEIVER

					&& choice != EXIT) {

			printf("\n NOT A VALIDE CHOICE! \n\n");
```

```
                fgets((char*) temp, MAX_STRING_SIZE, stdin);
                sscanf(temp, "%d", &choice);
        }

        return choice;
}

int start_settings() {
        settings.port = -1;
        settings.serialPortDescriptor = 0;
        settings.status = -1;
        settings.fileSize = 0;
        settings.packetSize = 0;
        settings.fileDescriptor = 0;
        settings.currentNum = 0;
        memset(settings.name, 0, PATH_MAX);

        return 0;
}
```

application.h

```
#ifndef APPLICATION_H_

#define APPLICATION_H_



#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <termios.h>

#include <stdio.h>

#include "link.h"

#include "definitions.h"


struct app_layer {

		int status;

		int port;

		int serialPortDescriptor;

		int fileDescriptor;

		int packetSize;

		int fileSize;

		unsigned char currentNum;

		unsigned char name[PATH_MAX];

}typedef app_layer_T;


int cliPort();

int cliStatus();

int startApp();

int setup();

int sendFile();

int start_settings();

int receiveFile();

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

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

int writeToFile(unsigned char * info, unsigned int byn_number);

int createFile(char * name);

int receiveControl(int var);

int sendPacket(unsigned char * message, unsigned int size);

int sendControl(int var);

int cliPacket();

int cliBaudrate();

int cliRetries();

char* cliChooseFile();

int getFileSize(char* path);

int validPath(unsigned char * path);

int cliTimeout();



#endif
```

definitions.h

```
#ifndef DEFINITIONS_H_

#define DEFINITIONS_H_


#include <linux/limits.h>

#include <stdint.h>

#define SENDER 1

#define RECEIVER 2

#define EXIT 3

#define PACKET_MAX_SIZE 65535

#define MAX_STRING_SIZE 10000


#define FLAG 0x7E

#define ADDR_TRANS 0x03

#define ADDR_REC_RESP 0x03

#define ADDR_REC 0x01

#define ADDR_TRANS_RESP 0x01

#define CTRL_SET 0x03

#define CTRL_UA 0x07

#define CTRL_DISC 0x0B


#define FALSE 0

#define TRUE 1


#define NEXT_INDEX(num) (num ^ 0x01)


#define OCTET_ESCAPE 0x7D

#define OCTET_DIFF 0x20


#define MAX_SIZE 256000


#define NEXT_CTRL_INDEX(num) (num << 1)

#define CTRL_REC_READY(num) ((num << 5) | 0x05)
```

```
#define CTRL_REC_REJECT(num) ((num << 1) | 0x01)


#define DISCONECTED -50


#endif
```

link.c

```
#include "link.h"

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <termios.h>

#include <stdio.h>

#include <stdlib.h>

#include <signal.h>

#include <unistd.h>

#include <string.h>

#include <strings.h>

#include "definitions.h"


enum frameFields {

	START_FLAG, ADDR, CTRL, BCC1, DATA, BCC2, END_FLAG

};


volatile int STOP=FALSE;


unsigned int initialized = 0;

unsigned int openLink = 0;


linkLayer_t linklayer;

unsigned int isLinkInit = 0;

unsigned int isLinkOpen = 0;

unsigned int isDefaultBR = 0;

unsigned int isDefaultTO = 0;

unsigned int isDefaultNT = 0;

unsigned int isDefaultHER = 0;

unsigned int isDefaultFER = 0;


struct termios oldtio, newtio;
```

```
int alarmFlag = 1;

unsigned int alarmCount = 0;


int init_link(unsigned int port);

int open_port_file(unsigned int port);

int close_port_file(int fd);

int send_set(int fd);

int rec_set(int fd);

int send_ua(int fd, unsigned int flag);

int send_disc(int fd, unsigned int flag);

int rec_disc(int fd, unsigned int flag);

void alarmListener();

int rec_ua(int fd, unsigned int flag);

int generate_bcc2(unsigned char * buffer, unsigned int length, unsigned char * bcc2);

int send_data(int fd, unsigned char * buffer, unsigned int length);

int byteStuffing(unsigned char * buffer, unsigned int length, unsigned char * new_buffer);

int rec_resp_receiver(int fd, unsigned char * buffer, unsigned int length, unsigned char bcc2);

int rec_data(int fd, unsigned char * buffer);

int byteDestuffing(unsigned char * buffer, unsigned int length, unsigned char * new_buffer);

int send_rej(int fd);

int send_rr(int fd);

int setBR(unsigned int baudrate);

int setTO(unsigned int timeout);

int setNT(unsigned int numtransmissions);

int setHER(unsigned int her);

int setFER(unsigned int fer);


int llopen(unsigned int port, unsigned int flag) {
        if (init_link(port))
                return -1;


        linklayer.flag = flag;
```

```
	int fd = open_port_file(port);


	if (flag == RECEIVER) {
		if (rec_set(fd))
			return -1;
		if (send_ua(fd, flag))
			return -1;
	} else if (flag == SENDER) {
		if (send_set(fd))
			return -1;
		if (rec_ua(fd, flag))
			return -1;
	} else {
		printf("ENTER A VALID STATUS!\n");
		close_port_file(fd);
		return -1;
	}


	initialized = -1;
	return fd;
}


int init_link(unsigned int port) {
	if (port<0 || port>4) {
		printf("ENTER A VALID PORT!\n");
		return -1;
	}


	sprintf(linklayer.port, "/dev/ttyS%d", port);


	if (!isDefaultBR)
		linklayer.baudRate = B38400;
	linklayer.sequenceNumber = 0;
```

```
	if (!isDefaultTO)
		linklayer.timeout = 3;
	if (!isDefaultNT)
		linklayer.numTransmissions = 3;
	linklayer.openLink = -1;
	if (!isDefaultHER)
		linklayer.her = 0;
	if (!isDefaultFER)
		linklayer.fer = 0;

	return 0;
}

int open_port_file(unsigned int port) {
	int fd;

	fd = open(linklayer.port, O_RDWR | O_NOCTTY);
	if (fd < 0) {
		perror(linklayer.port);
		close_port_file(fd);
		return -1;
	}

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

	bzero(&newtio, sizeof(newtio));
	newtio.c_cflag = linklayer.baudRate | CS8 | CLOCAL | CREAD;
	newtio.c_iflag = IGNPAR;
	newtio.c_oflag = OPOST;
```

```
	newtio.c_lflag = 0;


	newtio.c_cc[VTIME] = 0;
	newtio.c_cc[VMIN] = 1;


	tcflush(fd, TCIFLUSH);


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


	return fd;
}


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


	return 0;
}


int send_set(int fd) {
	unsigned char SET[5];


	SET[0] = FLAG;
	SET[1] = ADDR_TRANS;
	SET[2] = CTRL_SET;
```

```
	SET[3] = (SET[1] ^ SET[2]);
	SET[4] = FLAG;

	write(fd,SET,5);

	return 0;
}

int rec_set(int fd) {
	int i = START_FLAG;

	while (STOP==FALSE) {
		unsigned char c = 0;

		if (read(fd, &c,1)) {
			switch (i) {
				case START_FLAG:
					if (c == FLAG)
						i = ADDR;
					break;
				case ADDR:
					if (c == ADDR_TRANS) {
						i = CTRL;
					} else if (c != FLAG) {
						i = START_FLAG;
					}
					break;
				case CTRL:
					if (c == CTRL_SET) {
						i = BCC1;
					} else if (c == FLAG) {
						i = ADDR;
					} else {
```

```
                                        i = START_FLAG;
                                }
                                break;
                        case BCC1:
                                if (c == (ADDR_TRANS ^ CTRL_SET)) {
                                        i = END_FLAG;
                                } else if (c == FLAG) {
                                        i = ADDR;
                                } else {
                                        i = START_FLAG;
                                }
                                break;
                        case END_FLAG:
                                if (c == FLAG) {
                                        STOP = TRUE;
                                } else {
                                        i = START_FLAG;
                                }
                                break;
                        }
                }
        }

        return 0;
}

int send_ua(int fd, unsigned int flag) {
        unsigned char UA[5];

        UA[0] = FLAG;

        if (flag == SENDER) {
                UA[1] = ADDR_TRANS_RESP;
```

```
	} else if (flag == RECEIVER) {
		UA[1] = ADDR_REC_RESP;
	}

	UA[2] = CTRL_UA;
	UA[3] = UA[1] ^ UA[2];
	UA[4] = FLAG;

	write(fd, UA, 5);

	return 0;
}

int send_disc(int fd, unsigned int flag) {
	unsigned char DISC[5];

	DISC[0] = FLAG;

	if (flag == SENDER)
		DISC[1] = ADDR_TRANS;
	else if (flag == RECEIVER)
		DISC[1] = ADDR_REC_RESP;

	DISC[2] = CTRL_DISC;
	DISC[3] = DISC[1] ^ DISC[2];
	DISC[4] = FLAG;

	write(fd, DISC, 5);

	return 0;
}

void setAlarm() {
```

```
	struct sigaction sact;
	sact.sa_flags = 0;
	sact.sa_handler = alarmListener;
	sigaction(SIGALRM, &sact, NULL);
	alarm(linklayer.timeout);
	alarmFlag = 0;
	alarmCount = 0;
}

int rec_disc(int fd, unsigned int flag) {
	if (flag == SENDER) {
		setAlarm();
	}

	unsigned char addr = 0;
	unsigned char ctrl = 0;

	int i = START_FLAG;
	STOP = FALSE;

	while (STOP == FALSE) {
		unsigned char c = 0;

		if (flag == SENDER) {
			if (alarmCount >= linklayer.numTransmissions) {
				printf("EXCEDED NUMBER OF TRIES\n");
				close_port_file(fd);
				return -1;
			} else if (alarmFlag == 1) {
				printf("RE-SEND DISC!!!\n");
				send_disc(fd, flag);
				alarmFlag = 0;
				alarm(linklayer.timeout);
```

```
                }
        }

        if (read(fd, &c, 1)) {
                switch (i) {
                        case START_FLAG:
                                if (c == FLAG)
                                        i = ADDR;
                                break;
                        case ADDR:
                                if ((flag == SENDER && c == ADDR_REC_RESP) || (flag == RECEIVER && c ==
ADDR_TRANS)) {
                                        addr = c;
                                        i = CTRL;
                                } else if (c != FLAG) {
                                        i = START_FLAG;
                                }
                                break;
                        case CTRL:
                                if (c == CTRL_DISC) {
                                        ctrl = c;
                                        i = BCC1;
                                } else if (c == FLAG) {
                                        i = ADDR;
                                } else {
                                        i = START_FLAG;
                                }
                                break;
                        case BCC1:
                                if (c == (addr ^ ctrl)) {
                                        i = END_FLAG;
                                } else if (c == FLAG) {
                                        i = ADDR;
```

```
                                        } else {
                                                i = START_FLAG;
                                        }
                                        break;
                                case END_FLAG:
                                        if (c == FLAG) {
                                                STOP = TRUE;
                                        } else {
                                                i = START_FLAG;
                                        }
                                        break;
                                }
                        }
                }

                return 0;
}

void alarmListener() {
                printf("TIMEOUT!!!\n");
                alarmFlag = 1;
                alarmCount++;
}

int rec_ua(int fd, unsigned int flag) {
                setAlarm();

                unsigned int addr = 0;
                unsigned int ctrl = 0;

                int i = START_FLAG;
                STOP = FALSE;
```

```
while (STOP == FALSE) {
        unsigned char c = 0;

        if (alarmCount >= linklayer.numTransmissions) {
                printf("EXCEDED NUMBER OF TRIES!\n");
                close_port_file(fd);
                return -1;
        } else if (alarmFlag == 1) {
                printf("RE-SEND UA!!!\n");
                if (flag == SENDER) {
                        send_set(fd);
                } else if (flag == RECEIVER) {
                        send_disc(fd, flag);
                }

                alarmFlag = 0;
                alarm(linklayer.timeout);
        }
        if (read(fd, &c, 1)) {
                switch (i) {
                        case START_FLAG:
                                if (c == FLAG)
                                        i = ADDR;
                                break;
                        case ADDR:
                                if ((flag == SENDER && c == ADDR_REC_RESP) || (flag == RECEIVER && c ==
ADDR_TRANS_RESP)) {
                                        addr = c;
                                        i = CTRL;
                                } else if (c != FLAG) {
                                        i = START_FLAG;
                                }
                                break;
```

```
                                        case CTRL:
                                                if (c == CTRL_UA) {
                                                        ctrl = c;
                                                        i = BCC1;
                                                } else if (c == FLAG) {
                                                        i = ADDR;
                                                } else {
                                                        i = START_FLAG;
                                                }
                                                break;
                                        case BCC1:
                                                if (c == (addr ^ ctrl)) {
                                                        i = END_FLAG;
                                                } else if (c == FLAG) {
                                                        i = ADDR;
                                                } else {
                                                        i = START_FLAG;
                                                }
                                                break;
                                        case END_FLAG:
                                                if (c == FLAG) {
                                                        STOP = TRUE;
                                                } else {
                                                        i = START_FLAG;
                                                }
                                                break;
                                }
                        }
                }
                return 0;
}


int llwrite(int fd, unsigned char * buffer, unsigned int length) {
```

```
if (linklayer.flag == RECEIVER) {
		printf("RECEIVER CANNOT WRITE TO SERIAL PORT\n");
		return -1;
}

if (!initialized) {
		printf("SERIAL PORT ISNT INITIALIZED\n");
		return -1;
}
unsigned char bcc2 = 0;
generate_bcc2(buffer, length, &bcc2);

unsigned char aux_buffer[MAX_SIZE] = "";
memcpy(aux_buffer, buffer, length);
printf("bcc2 wrote: %x\n",bcc2);
aux_buffer[length] = bcc2;

unsigned char * stuffed_buffer = malloc(MAX_SIZE);
int stuffed_length = 0;

if ((stuffed_length = byteStuffing(aux_buffer, length+1, stuffed_buffer)) == -1) {
		printf("BYTE STUFFING ERROR\n");
		return -1;
}
int nbytes = 0;
printf("sending data!\n");
nbytes = send_data(fd, stuffed_buffer, stuffed_length);
printf("wrote: %d bytes",nbytes);
if (rec_resp_receiver(fd, stuffed_buffer, stuffed_length, bcc2) == -1) {
		printf("\n");
		return -1;
}
```

```
	free(stuffed_buffer);

	linklayer.sequenceNumber = NEXT_INDEX(linklayer.sequenceNumber);


	return nbytes;
}


int generate_bcc2(unsigned char * buffer, unsigned int length, unsigned char * bcc2) {


	*bcc2 = 0;

	unsigned int i;


	for (i=0 ; i<length ; i++) {

	*bcc2 ^= buffer[i];

	}


	return 0;
}


int byteStuffing(unsigned char * buffer, unsigned int length, unsigned char * new_buffer) {

	unsigned int buff_pos = 0;

	unsigned int i;


	for (i=0 ; i<length ; i++) {

		char c = buffer[i];


		if (c == FLAG || c == OCTET_ESCAPE) {

			new_buffer[buff_pos++] = OCTET_ESCAPE;

			new_buffer[buff_pos++] = c ^ OCTET_DIFF;

		} else {

			new_buffer[buff_pos++] = c;

		}

//		printf("stuffed a %x to a %x \n",buffer[i],c);

	}
```

```
	return buff_pos;
}

int send_data(int fd, unsigned char * buffer, unsigned int length) {
	unsigned char flag = FLAG;
	unsigned char addr = ADDR_TRANS;
	unsigned char ctrl = NEXT_CTRL_INDEX(linklayer.sequenceNumber);
	unsigned char bcc1 = addr ^ ctrl;

	printf("writing!!\n");
	write(fd, &flag, 1);
	write(fd, &addr, 1);
	write(fd, &ctrl, 1);
	write(fd, &bcc1, 1);
	write(fd, buffer, length);
	write(fd, &flag, 1);
	printf("sender wrote: %x %x %x %x data %x\n",flag,addr,ctrl,bcc1,flag);
	/*printf("data: ");
	int indice;
	for(indice=0;indice<length;indice++) {
		printf("%x",buffer[indice]);
	}
	printf("\n");*/
	return (length+5);
}

int rec_resp_receiver(int fd, unsigned char * buffer, unsigned int length, unsigned char bcc2) {
	setAlarm();

	unsigned char addr = 0;
	unsigned char ctrl = 0;
```

```
int i = START_FLAG;

STOP = FALSE;


while (STOP == FALSE) {
        unsigned char c = 0;


        if (alarmCount >= linklayer.numTransmissions) {
                printf("EXCEDED NUMBER OF TRIES\n");
                close_port_file(fd);
                return -1;
        } else if (alarmFlag == 1) {
                printf("RE-SEND RR!!!\n");
                send_data(fd, buffer, length);
                alarmFlag = 0;
                alarm(linklayer.timeout);
        }


        if (read(fd, &c, 1)) {
                switch (i) {
                        case START_FLAG:
                                if (c == FLAG)
                                        i = ADDR;
                                break;
                        case ADDR:
                                if (c == ADDR_REC_RESP) {
                                        addr = c;
                                        i = CTRL;
                                } else if (c != FLAG) {
                                        i = START_FLAG;
                                }
                                break;
                        case CTRL:
```

```
                                        if ((c == CTRL_REC_READY(NEXT_INDEX(linklayer.sequenceNumber))) || (c
== CTRL_REC_READY(linklayer.sequenceNumber)) || (c == CTRL_REC_REJECT(linklayer.sequenceNumber))) {
                                                ctrl = c;
                                                i = BCC1;
                                        } else if (c == FLAG) {
                                                i = ADDR;
                                        } else {
                                                i = START_FLAG;
                                        }
                                        break;
                                case BCC1:
                                        if (c == (addr ^ ctrl)) {
                                                i = END_FLAG;
                                        } else if (c == FLAG) {
                                                i = ADDR;
                                        } else {
                                                i = START_FLAG;
                                        }
                                        break;
                                case END_FLAG:
                                        if (c == FLAG) {
                                                if (ctrl ==
CTRL_REC_READY(NEXT_INDEX(linklayer.sequenceNumber))) {
                                                        alarm(0);
                                                        STOP = TRUE;
                                                } else if (ctrl == CTRL_REC_REJECT(linklayer.sequenceNumber)) {
                                                        send_data(fd, buffer, length);
                                                        i = START_FLAG;
                                                        alarm(linklayer.timeout);
                                                } else if (ctrl == CTRL_REC_READY(linklayer.sequenceNumber)) {
                                                        alarm(0);
                                                        STOP = TRUE;
                                                } else {
                                                        i = ADDR;
```

```
                                                }
                                        } else {
                                                i = START_FLAG;
                                        }
                                        break;
                                }
                        }
                }

                return 0;
}

int llread(int fd, unsigned char * buffer) {
                if (linklayer.flag == SENDER) {
                        printf("SENDER CANNOT READ FROM SERIAL PORT");
                        return -1;
                }

                if (!initialized) {
                        printf("SERIAL PORT ISNT INITIALIZED\n");
                        return -1;
                }

                return rec_data(fd, buffer);
}

int rec_data(int fd, unsigned char * buffer) {
                unsigned char addr = 0;
                unsigned char ctrl = 0;
                unsigned int dataCount = 0;
                unsigned char stuffed_buffer[MAX_SIZE];
                int i = START_FLAG;
                STOP = FALSE;
```

```
while (STOP == FALSE) {
        unsigned char c = 0;
        //printf("start while \n");
        if (read(fd, &c, 1)) {
                //printf("received a: %x\n",c);
                switch (i) {
                        case START_FLAG:
                                //printf("start\n");
                                if (c == FLAG)
                                        i = ADDR;
                                break;
                        case ADDR:
//printf("adr\n");
                                if (c == ADDR_TRANS) {
                                        addr = c;
                                        i = CTRL;
                                } else if (c != FLAG) {
                                        i = START_FLAG;
                                }
                                break;
                        case CTRL:
                                //printf("ctrl\n");
                                if (c == NEXT_CTRL_INDEX(linklayer.sequenceNumber) || c == CTRL_DISC) {
                                        ctrl = c;
                                        i = BCC1;
                                } else if (c == FLAG) {
                                        i = ADDR;
                                } else {
                                        i = FLAG;
                                }
                                break;
                        case BCC1:
```

```
                                        //printf("bcc1\n");
                                        ;
                                        int headerErrorProb = rand() % 100;

                                        if (headerErrorProb < linklayer.her) {
                                                i = START_FLAG;
                                        } else {
                                                if (c == (addr ^ ctrl)) {
                                                        i = DATA;
                                                } else if (c == FLAG) {
                                                        i = ADDR;
                                                } else {
                                                        i = START_FLAG;
                                                }
                                        }
                                        break;
                                case DATA:
                                        //printf("data\n");
                                        if (c != FLAG) {

                                                stuffed_buffer[dataCount] = c;
//printf("stuffed: %x\n",stuffed_buffer[dataCount]);
                                                dataCount++;
                                        } else {
//printf("else\n");
                                                if (ctrl == CTRL_DISC) {
                                                        printf("send disc\n");
                                                        if (send_disc(fd, RECEIVER))
                                                                return -1;
                                                        printf("rec ua\n");
                                                        if (rec_ua(fd, RECEIVER))
                                                                return -1;
```

```
                linklayer.openLink = 0;
                return DISCONECTED;
            } else {
                //printf("else data lenght\n");
                unsigned int dataLength = 0;

                //printf("data count: %d",dataCount);
                if ((dataLength = byteDestuffing(stuffed_buffer,
dataCount, buffer)) == -1)
                    return -1;

                unsigned char bcc2_received = buffer[dataLength - 1];
                //printf("bcc2 received: %x\n", bcc2_received);
                unsigned char bcc2 = 0;
                //printf("bcc2\n");

                generate_bcc2(buffer, dataLength - 1, &bcc2);
                //printf("bcc generated: %x\n",bcc2);
                int frameErrorProb = rand() % 100;

                if (frameErrorProb < linklayer.fer) {
                    printf("send rej\n");
                    send_rej(fd);
                    i = START_FLAG;
                    dataCount = 0;
                } else {
                    //printf("else bcc2\n");
                    if (bcc2 == bcc2_received) {
                        printf("same bcc2\n");
                        if (ctrl !=
NEXT_CTRL_INDEX(linklayer.sequenceNumber)) {
                            printf("send rr1\n");
                            send_rr(fd);
                            i = START_FLAG;
```

```
                                                                dataCount = 0;
                                                        } else {
                                                                printf("send rr 2\n");

        linklayer.sequenceNumber = NEXT_INDEX(linklayer.sequenceNumber);
                                                                send_rr(fd);

                                                                return (dataLength -
1);
                                                        }
                                                } else {
                                                        printf("not the same bcc2\n");
                                                        if (ctrl !=
NEXT_CTRL_INDEX(linklayer.sequenceNumber))
                                                                send_rr(fd);
                                                        else
                                                                send_rej(fd);

                                                        i = START_FLAG;
                                                        dataCount = 0;
                                                }
                                        }
                                }
                        }
                        break;
                }
        }
        }
        printf("return\n");
        return 0;
}

int byteDestuffing(unsigned char * buffer, unsigned int length, unsigned char * new_buffer) {
        unsigned int destuff_pos = 0;
```

```
	unsigned int i = 0;
	//printf("started destuffing lenght: %d\n",length);
	for (; i<length ; i++) {
		//printf("buffer\n");
		char c = buffer[i];
		if (c == OCTET_ESCAPE) {
			c = buffer[++i];

			if (c == (FLAG ^ OCTET_DIFF)) {
				new_buffer[destuff_pos++] = FLAG;
			} else if (OCTET_ESCAPE ^ OCTET_DIFF) {
				new_buffer[destuff_pos++] = OCTET_ESCAPE;
			} else {
				printf("DESTUFFING BUFFER ERROR\n");
				return -1;
			}
		} else {
			new_buffer[destuff_pos++] = c;
		}
	//printf("destuffed a %x to a %x\n",buffer[i],c);
}

	return destuff_pos;
}

int send_rej(int fd) {
	unsigned char REJ[5];

	REJ[0] = FLAG;
	REJ[1] = ADDR_REC_RESP;
	REJ[2] = CTRL_REC_REJECT(linklayer.sequenceNumber);
	REJ[3] = REJ[1] ^ REJ[2];
	REJ[4] = FLAG;
```

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


		return 0;
}


int send_rr(int fd) {
		unsigned char RR[5];


		RR[0] = FLAG;
		RR[1] = ADDR_REC_RESP;
		RR[2] = CTRL_REC_READY(linklayer.sequenceNumber);
		RR[3] = RR[1] ^ RR[2];
		RR[4] = FLAG;


		write(fd, RR, 5);


		return 0;
}


int llclose(int fd, unsigned int flag) {
		if (!initialized) {
				printf("THIS IS ALREADY CLOSE\n");
				return -1;
		}


		if (flag == SENDER) {
				if (send_disc(fd, flag))
						return -1;
				if (rec_disc(fd, flag))
						return -1;
				if (send_ua(fd, flag))
						return -1;
```

```
        }

        if (close_port_file(fd))
                return -1;

        return 0;
}

int setBR(unsigned int baudrate) {
        linklayer.baudRate = baudrate;
        isDefaultBR = -1;

        return 0;
}

int setTO(unsigned int timeout) {
        linklayer.timeout = timeout;
        isDefaultTO = -1;

        return 0;
}

int setNT(unsigned int numtransmissions) {
        linklayer.numTransmissions = numtransmissions;
        isDefaultNT = -1;

        return 0;
}

int setHER(unsigned int her) {
        linklayer.her = her;
        isDefaultHER = -1;
```

```
        return 0;
}

int setFER(unsigned int fer) {
        linklayer.fer = fer;
        isDefaultFER = -1;

        return 0;
}
```

link.h

```
#ifndef _LINK_H_

#define _LINK_H_


typedef struct linkLayer {

        char port[20];

        unsigned int baudRate;

        unsigned int sequenceNumber;

        unsigned int timeout;

        unsigned int numTransmissions;

        unsigned int openLink;

        int flag;

        unsigned int her;

        unsigned int fer;

} linkLayer_t;


void setAlarm();

int llopen(unsigned int port, unsigned int flag);

int llwrite(int fd, unsigned char * buffer, unsigned int length);

int llread(int fd, unsigned char * buffer);

int llclose(int fd, unsigned int flag);

#endif
```

main.c

```
#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <termios.h>

#include <stdio.h>

#include "application.h"


int main(int argc, char argv[]) {


	startApp();


	return 1;

}
```

Makefile

```
all:

	gcc main.c application.c link.c -o  app
```