Universidade do Porto

FEUP Faculdade de Engenharia

Relatório do 1º Trabalho Laboratorial
Protocolo de Ligação de Dados

Mestrado Integrado em Engenharia Informática e Computação
3º Ano

Redes de Computadores
2016/2017

Alexandre Saraiva Moreira, up201303281@fe.up.pt
Paulo Renato Almeida Correia, up201406006@fe.up.pt
Paulo Jorge Silva Ferreira, up201305617@fe.up.pt

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, 4200–65 Porto, Portugal

Novembro 2016

Índice

Sumário

No âmbito da unidade curricular de Redes de Computadores, do curso Mestrado Integrado em Engenharia Informática e Computação, foi-nos proposto a elaboração de uma aplicação cujo objectivo era a transferência de ficheiros entre dois computadores através de uma ligação por cabo pelas portas série de cada um.

A elaboração de um relatório final para o trabalho tem como objetivo a consolidação do trabalho realizado ao longo desta primeira metade de semestre. Neste contexto pode-se afirmar que é necessário que haja uma ligação entre a parte prática e teórica que envolveu o projeto.

Introdução

O trabalho realizado ao longo das aulas laboratoriais da unidade curricular de Redes de Computadores tem como objetivo a implementação um protocolo de ligação de dados, de acordo com a especificação descrita no guião.

Na sua especificação era pedida a combinação de características de protocolos de ligação de dados existentes, entre os quais a transparência na transmissão de dados e uma transmissão organizada em diferentes tipos de tramas – tais como tramas de informação, supervisão e não numeradas, que eram protegidas por código detetor de erros.

Outro objetivo do trabalho consistia em testar o protocolo com uma aplicação simples de transferência de ficheiros, igualmente especificada. Na especificação da aplicação de teste era pedido o suporte de dois tipos de pacotes enviados pelo emissor: o de dados e o de controlo.

Nos pacotes de controlo destacam-se o de start, sinalizando o início da transmissão; e o de end, sinalizando o fim da transmissão. Já os pacotes de dados iriam conter frações do ficheiro a transmitir.

A Introdução servirá, também, para dar indicações sobre o tipo de informação que poderá ser encontrada em cada uma das secções seguintes. Assim sendo, o relatório subdivide-se nas seguintes partes:

- Introdução - indicação dos objectivos do trabalho e do relatório
- Arquitectura - blocos funcionais e interfaces
- Estrutura do código - APIs, principais estruturas de dados, principais funções e a sua relação com a Arquitectura
- Casos de uso principais - fazer a sua identificação e abordar as sequências de chamada de funções
- Protocolo de ligação lógica - descrição da estratégia de implementação dos principais aspectos funcionais
- Protocolo de aplicação - descrição da estratégia de implementação dos principais aspectos funcionais
- Validação - descrição dos testes efectuados
- Elementos de valorização - descrição da estratégia de implementação dos elementos de valorização implementados
- Conclusões - onde estará presente o comentário e análise final do grupo perante este projeto

Arquitectura

Camada de aplicação e de ligação de dados

O trabalho está organizado em duas camadas – layers – que permitem a correta funcionalidade do projeto: a camada do protocolo de ligação de dados e a camada de aplicação. O ficheiro Datalink.h representa a camada de ligação de dados enquanto o ficheiro Application.h representa a camada de aplicação.

A camada de ligação de dados disponibiliza funções genéricas do protocolo, estando nela especificadas as funcionalidades de sincronismo e numeração de tramas, suporte de ligação, controlo da transferência, de erros e de fluxo, e todos os aspetos relacionados com a ligação à porta série – tais como a sua abertura e definição das suas propriedades.

A camada de aplicação tira proveito das propriedades da camada de ligação de dados para a implementação das suas funções. Podemos dizer que a camada de aplicação é a responsável pela transferência dos ficheiros pois é nesta que são executadas as funções de receção e emissão de tramas. Os pacotes de controlo e de dados também são processados e enviados a partir desta camada.

Interface da linha de comandos

Para colocar o programa em execução é necessário primeiramente compilar, para isso é preciso utilizar o seguinte comando: gcc Main.c -o “Nome_do_executável”. De seguida, para executar a aplicação utilizar o comando: ./”Nome_do_executável” <mode> <port> <file>.

Onde:

1. <mode> pode ter dois valores diferentes:
    - “-s” caso queira que o programa funcione como emissor;
    - “-r” caso pretenda que se comporte como receptor.

2. <port> também pode variar entre dois valores:
    - “/dev/ttyS0”;
    - “/dev/ttyS1”.

3. <file> é o ficheiro que pretende utilizar. Uma vez que o programa não transmite o nome do ficheiro, caso o programa vá desempenhar o papel de emissor o <file> deve referir-se a um ficheiro que exista e que pretenda que seja transmitido. Se por outro lado o programa se comportar como receptor o <file> pode ser o nome que pretende que o ficheiro possua quando por recebido pelo receptor.

Estrutura do código

Application layer

No ficheiro Application.h encontra-se a implementação da camada de aplicação e as suas funções principais. Para tal, utiliza-se uma estrutura de dados onde é guardado o descritor de ficheiro da porta série, o modo de ligação selecionado, o nome de ficheiro do ficheiro a ser recebido/enviado e, por fim, o identificador da porta selecionada.

```
54  typedef struct {
55      int fd, mode;
56      char *port, *fileName;
57  } Application;
```

As funções principais da camada encontram-se descritas na figura em baixo.

```
13 › int sendControlPackage(int ctrl, unsigned char *buffer, int length) {⋯
34
35 › int sendDataPackage(int N, unsigned char *buffer, int length) {⋯
56
57 › int receiveControlPackage(int ctrl, long int *size) {⋯
78
79 › int receiveDataPackage(unsigned char *buf) {⋯
95
96 › int sendFile(FILE *file) {⋯
135
136 › int receiveFile(FILE *file) {⋯
166
167 › int initApplication(int mode, char *port, char *fileName) {⋯
```

Data Link layer

No ficheiro Datalink.h encontra-se a implementação da camada de ligação de dados e as suas funções principais. Para tal, utiliza-se um estrutura de dados onde é guardado o número de tentativas antes de abortar o programa, o valor do tempo a ser esperado antes da próxima tentativa e as estruturas do termios usado pelo sistema operativo e o termios usado pelo nosso programa.

```
59  typedef struct {
60      int ns, retries, timeout;
61      struct termios oldtio, newtio;
62  } DataLink;
```

As funções principais de camada encontram-se descritas na figura em baixo.

```
12 › int initDataLink() {…
42
43 › unsigned char *createCommand(int command) {…
72
73 › unsigned char *createFrame(unsigned char *buffer, int length) {…
95
96 › int llopen(int fd, int mode) {…
162
163 › int llwrite(int fd, unsigned char *buffer, int length) {…
204
205 › int llread(int fd, unsigned char **buffer) {…
335
336 › int llclose(int fd, int mode) {…
424
425 › int closeSerialPort() {…
```

Casos de uso principais

A aplicação contém um caso de uso principal, sendo este a capacidade de transferência de ficheiros entre dois computadores através da porta série. Neste caso, o programa divide-se em duas vertentes. Na de ele estar a ser executado como Emissor ou Receptor.

Na primeira situação, as funções mais importantes são:

- sendFile
- sendControlPackage - dá uso à função llwrite
- sendDataPackage - dá uso à função llwrite

Na segunda situação, são as seguintes:

- receiveFile
- receiveControlPackage - dá uso à função llread
- receiveDataPackage - dá uso à função llread

Ambas as situações dão uso a funções fundamentais ao funcionamento do programa - llopen e llclose.

Protocolo de ligação lógica

A camada de ligação de dados é responsável pelas seguintes funcionalidades:

- Estabelecer e terminar uma ligação
- Criar e enviar comandos e tramas
- Receber e processar comandos e tramas
- Fazer *stuff* e *destuff* das tramas criadas

API da porta série

As funções llopen, llwrite, llread e llclose constituem a API da porta série e estão implementadas na camada de ligação de dados.

llopen

É a função responsável pelo estabelecimento da ligação através da porta série. Assim que a função é chamada envia um comando SET, e fica a espera de um comando UA, como resposta, para que possa prosseguir. Caso não receba a resposta, o programa tenta de novo um número pré-estabelecido de vezes. Se ainda assim não receber o UA o programa termina.

llwrite

Esta função recebe um buffer, o tamanho desse buffer e um identificador da ligação de dados, e tenta enviar esse buffer através da porta série e fica à espera de uma resposta. Se a resposta não chegar ao fim de um tempo pré-definido, é feita uma nova tentativa. Se receber uma resposta e esta for RR, é indicativo de que o buffer foi transmitido corretamente, se, pelo contrário, a resposta for REJ, indica que o buffer não foi transmitido corretamente e é feita uma nova tentativa.

llread

Esta função recebe um identificador da ligação de dados e um buffer e é responsável por receber dados através da porta série. Envia o comando REJ em resposta à função llwrite, se o buffer não foi recebido corretamente e envia um RR em caso de sucesso na recepção do buffer.

llclose

Esta função recebe um identificador de dados e é responsável por terminar a ligação através da porta série. Envia um comando DISC para o receptor indicando o fim da transmissão e recebe outro DISC do receptor como resposta. Por fim envia um comando UA para o receptor.

Protocolo de aplicação

O protocolo de aplicação é responsável pelos pacotes de controlo, pacotes de dados e pela recepção e transmissão do ficheiro. Está implementada no ficheiro *Application.h*. Os pacotes são diferenciados pelo campo de controlo presentes no início de cada pacote.

Os pacotes de controlo marcam o início e o fim da transmissão, se o valor do campo de controlo for dois significa que se trata do início da transmissão, se, pelo contrário, o valor for três trata-se do fim da transmissão. É também nos pacotes de controlo que são enviadas informações do ficheiro, como por exemplo, o tamanho e, opcionalmente, o nome.

```
38  #define CTRL_PKG_DATA    1
39  #define CTRL_PKG_START   2
40  #define CTRL_PKG_END     3
```

Os pacotes de dados têm como valor no campo de controlo o número um e transportam os segmentos do ficheiro a ser transmitido.

A transmissão de um ficheiro processa-se do seguinte modo:

- Inicialmente é enviado o pacote de controlo de início de transmissão, onde vão informações como o tamanho do ficheiro
- De seguida são enviados os pacotes de dados até todo o ficheiro ter sido completamente transferido
- Por fim é enviado o pacote de controlo que marca o fim de transmissão. Este último pacote é exatamente igual ao pacote de inicio de transmissão, à excepção do campo de controlo

Validação

Para testar se o programa funcionava de acordo com as especificações, foram utilizadas diferentes imagens com diferentes tamanhos e também um ficheiro de texto. Todas estas transferências de teste ocorreram conforme o previsto.

Também foi testada a capacidade da aplicação funcionar mesmo que sejam provocadas interrupções na ligação da porta série e/ou provocados curto-circuitos e/ou outro de tipo de imprevistos que levassem a transferência de “lixo” em vez dos pacotes de dados criados na camada de aplicação. O programa reconheceu e recuperou de tais circunstâncias.

Elementos de valorização

Implementação de REJ

Caso seja detectado um erro no BCC1 ou no BCC2 na função llread, é criado e enviado um comando REJ para o emissor para que a trama com o erro seja reenviada.

```
COMMAND = createCommand(REJ);
stats->rejSent++;

if (write(fd, COMMAND, COMMAND_SIZE) == -1) {
    printf("ERROR: Failed to send REJ buffer.\n");
    return -1;
}
```

Verificação da integridade dos dados

A aplicação verifica se o tamanho do ficheiro recebido é igual ao tamanho do ficheiro enviado. Os pacotes são numerados para garantir que não são enviados pacotes repetidos.

Registo de ocorrências

Ao longo da execução do programa, vão sendo registadas as várias ocorrências de *timeouts*, os REJs enviados e recebidos e as tramas enviadas, recebidas ou reenviadas. Estes dados são guardados na *struct* Statistics.

```
64  typedef struct {
65      int numberTimeout;
66      int rejSent, rejReceived;
67      int frameSent, frameResent, frameReceived;
68  } Statistics;
```

Conclusões

A realização deste projeto contribuiu para a consolidação dos conceitos interiorizados nas aulas teóricas e laboratoriais, para um conhecimento mais profundo de como a comunicação em redes funciona e até para um uso da porta série.

Em geral, sentimos que todos os elementos do grupo terminaram com uma compreensão bem clara do funcionamento do projeto implementado e todos concordam que a maior dificuldade esteve, não na implementação do código, mas sim na compreensão de como é que tudo deveria funcionar e se interligar, em termos de criação e envio de pacotes de controlo, pacotes de dados, detecção e recuperação de erros, entre outros.

Anexo

Main.c

```
#include <stdio.h>
#include <string.h>
#include "Application.h"

void printUsage(char *command) {
	printf("Usage: %s <mode> <port> <file>\n", command);
	printf("Where <mode> includes:\n");
	printf("    -s    Send a file\n");
	printf("    -r    Receive a file\n");
	printf("And <port> includes:\n");
	printf("    /dev/ttyS0\n");
	printf("    /dev/ttyS1\n");
}

int processArguments(char **argv) {
	int mode;
	char *port, *fileName;

	if (strcmp(argv[1], "-s") == 0)
		mode = SEND;
	else if (strcmp(argv[1], "-r") == 0)
		mode = RECEIVE;
	else {
		printf("ERROR: Neither send or receive was specified.\n");
		printUsage(argv[0]);
		return -1;
	}

	if (strcmp(argv[2], "/dev/ttyS0") == 0 || strcmp(argv[2],
"/dev/ttyS1") == 0)
		port = argv[2];
	else {
		printf("ERROR: Please choose a valid port.\n");
		printUsage(argv[0]);
		return -1;
	}

	if (mode == SEND) {
		FILE *file;
		file = fopen(argv[3], "r");

		if (file == NULL) {
			printf("ERROR: File \"%s\" does not exist.\n", argv[3]);
			printUsage(argv[0]);
			return -1;
		} else {
			fclose(file);
			fileName = argv[3];
		}
	} else if (mode == RECEIVE) {
		fileName = argv[3];
	}

	initApplication(mode, port, fileName);
```

```
	return 0;
}

int main(int argc, char **argv) {
	if (argc != 4) {
		printf("ERROR: Wrong number of arguments.\n");
		printUsage(argv[0]);
		return -1;
	} else {
		processArguments(argv);
	}

	return 0;
}
```

Application.h

```
#ifndef APPLICATION_H
#define APPLICATION_H

#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include "Utils.h"
#include "DataLink.h"
#include "ByteStuffing.h"

int sendControlPackage(int ctrl, unsigned char *buffer, int length) {
	int i;
	unsigned char ctrlPackage[CTRL_PKG_SIZE];

	ctrlPackage[0] = ctrl;
	ctrlPackage[1] = PARAM_FILE_SIZE;
	ctrlPackage[2] = length;

	for (i = 0; i < length; i++)
		ctrlPackage[i + 3] = buffer[i];

	if (llwrite(app->fd, ctrlPackage, CTRL_PKG_SIZE) == -1)
		return -1;

	if (ctrl == CTRL_PKG_START)
		printf("Sent START control package.\n");
	else if (ctrl == CTRL_PKG_END)
		printf("\nSent END control package.\n");

	return 0;
}

int sendDataPackage(int N, unsigned char *buffer, int length) {
	int i;
	unsigned char dataPackage[length + DATA_PKG_SIZE];

	dataPackage[0] = CTRL_PKG_DATA;
```

```
	dataPackage[1] = N % 255;
	dataPackage[2] = length / 256;
	dataPackage[3] = length % 256;

	for (i = 0; i < length; i++)
		dataPackage[i + 4] = buffer[i];

	int aux = llwrite(app->fd, dataPackage, length + DATA_PKG_SIZE);

	if (aux == -1)
		return -1;
	else if(aux == -2)
		return -2;

	return 0;
}

int receiveControlPackage(int ctrl, long int *size) {
	unsigned char *answer;

	if (llread(app->fd, &answer) == -1)
		return -1;

	if (answer[4] == CTRL_PKG_START)
		printf("Received START control package.\n");
	else if (answer[4] == CTRL_PKG_END)
		printf("\nReceived END control package.\n");
	else
		return -1;

	int i;
	unsigned char fileSizeBuf[8];
	for (i = 0; i < 8; i++)
		fileSizeBuf[i] = answer[i + 7];
	memcpy(size, fileSizeBuf, 8);

	return 0;
}

int receiveDataPackage(unsigned char *buf) {
	int i, readBytes = 0;
	unsigned char *package;

	readBytes = llread(app->fd, &package);
	if (readBytes == -1)
		return -1;
	else if (readBytes == -2) {
		return -2;
	}

	for (i = 0; i < readBytes; i++)
		buf[i] = package[i + 8];

	return readBytes;
}

int sendFile(FILE *file) {
	long int fileSize = getFileSize(app->fileName);
	unsigned char *fileSizeBuf = (unsigned char *)&fileSize;
```

```
	int aux;

	if (sendControlPackage(CTRL_PKG_START, fileSizeBuf,
sizeof(fileSizeBuf)) == -1) {
		printf("ERROR: Failed to send the START control package.\n");
		return -1;
	}

	unsigned char fileBuf[MAX_SIZE];
	size_t readBytes = 0, writtenBytes = 0, i = 0;

	while ((readBytes = fread(fileBuf, sizeof(unsigned char), MAX_SIZE,
file)) > 0) {
		aux = sendDataPackage(i++, fileBuf, readBytes);
		if (aux == -1) {
			printf("ERROR: Failed to send one of the DATA
packages.\n");
			return -1;
		} else if(aux == -2) {
			while(sendDataPackage(i++, fileBuf, readBytes) == -2) {
				stats->frameResent++;
				continue;
			}
		}
		stats->frameSent++;

		memset(fileBuf, 0, MAX_SIZE);
		writtenBytes += readBytes;

		printProgress(writtenBytes, fileSize);
	}

	if (sendControlPackage(CTRL_PKG_END, fileSizeBuf,
sizeof(fileSizeBuf)) == -1) {
		printf("ERROR: Failed to send the END control package.\n");
		return -1;
	}

	return 0;
}

int receiveFile(FILE *file) {
	long int fileSize;
	if (receiveControlPackage(CTRL_PKG_START, &fileSize) == -1) {
		printf("ERROR: Failed to receive the START control
package.\n");
		return -1;
	}

	unsigned char tempBuf[MAX_SIZE];
	size_t readBytes = 0, receivedBytes = 0, i = 0;
	while (readBytes < fileSize) {
		if ((receivedBytes = receiveDataPackage(tempBuf)) == -2) {
			memset(tempBuf, 0, MAX_SIZE);
			continue;
		}

		stats->frameReceived++;
		fwrite(tempBuf, 1, receivedBytes, file);
		memset(tempBuf, 0, MAX_SIZE);
```

```
            readBytes += receivedBytes;
            printProgress(readBytes, fileSize);
      }

      if (receiveControlPackage(CTRL_PKG_END, &fileSize) == -1) {
            printf("\nERROR: Failed to receive the END control
package.\n");
            return -1;
      }

      return 0;
}

int initApplication(int mode, char *port, char *fileName) {
      app = (Application *)malloc(sizeof(Application));
      stats = (Statistics *)malloc(sizeof(Statistics));

      app->mode = mode;
      app->port = port;
      app->fileName = fileName;
      app->fd = open(port, O_RDWR | O_NOCTTY);

      stats->numberTimeout = 0;
      stats->rejSent = 0;
      stats->rejReceived = 0;
      stats->frameSent = 0;
      stats->frameResent = 0;
      stats->frameReceived = 0;

      if (app->fd < 0) {
            printf("ERROR: Failed to open serial port.\n");
            return -1;
      }

      initDataLink();

      if (llopen(app->fd, app->mode) == -1) {
            printf("ERROR: Failed to create a connection.\n");
            return -1;
      } else {
            printf("Connection successful.\n");
      }

      if (app->mode == SEND) {
            FILE *file;
            file = fopen(app->fileName, "rb");
            if (file == NULL) {
                  printf("ERROR: Failed to open file \"%s\".\n", app-
>fileName);
                  return -1;
            }

            if (sendFile(file) == -1) {
                  printf("ERROR: Failed to send file \"%s\".\n", app-
>fileName);
                  return -1;
            }

            if (fclose(file) != 0) {
```

```
                printf("ERROR: Failed to close file \"%s\".\n", app-
>fileName);
                return -1;
            }
      } else if (app->mode == RECEIVE) {
            FILE *file;
            file = fopen(app->fileName, "wb");
            if (file == NULL) {
                printf("ERROR: Failed to create file \"%s\".\n", app-
>fileName);
                return -1;
            }

            if (receiveFile(file) == -1) {
                printf("ERROR: Failed to receive file \"%s\".\n", app-
>fileName);
                return -1;
            }

            if (fclose(file) != 0) {
                printf("ERROR: Failed to close file \"%s\".\n", app-
>fileName);
                return -1;
            }
      }

      if (llclose(app->fd, app->mode) == -1) {
            printf("ERROR: Failed to close the connection.\n");
            return -1;
      } else {
            printf("Disconnection successful.\n");
      }

      closeSerialPort();
      printStatistics();
}

#endif
```

Datalink.h

```
#ifndef DATALINK_H
#define DATALINK_H

#include <stdio.h>
#include <stdlib.h>
#include "Utils.h"
#include "Alarm.h"
#include "ByteStuffing.h"

volatile int STOP = FALSE;

int initDataLink() {
      dl = (DataLink *)malloc(sizeof(DataLink));

      dl->ns = 0;
      dl->retries = 3;
      dl->timeout = 3;
```

```
	// Save current port settings
	if (tcgetattr(app->fd, &dl->oldtio) == -1) {
		printf("ERROR: Could not save current port settings.\n");
		return -1;
	}

	// Set new termios structure
	bzero(&dl->newtio, sizeof(dl->newtio));
	dl->newtio.c_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;
	dl->newtio.c_iflag = IGNPAR;
	dl->newtio.c_oflag = 0;
	dl->newtio.c_lflag = 0;
	dl->newtio.c_cc[VMIN] = 1;
	dl->newtio.c_cc[VTIME] = 0;

	tcflush(app->fd, TCIOFLUSH);
	if (tcsetattr(app->fd, TCSANOW, &dl->newtio) == -1) {
		printf("ERROR: Failed to set new termios structure.\n");
		return -1;
	}

	return 0;
}

unsigned char *createCommand(int command) {
	unsigned char *buf = (unsigned char *)malloc(COMMAND_SIZE);

	buf[0] = FLAG;
	buf[1] = A;

	switch (command) {
	case UA:
		buf[2] = C_UA;
		break;
	case SET:
		buf[2] = C_SET;
		break;
	case DISC:
		buf[2] = C_DISC;
		break;
	case RR:
		buf[2] = C_RR;
		break;
	case REJ:
		buf[2] = C_REJ;
		break;
	}

	buf[3] = buf[1] ^ buf[2];
	buf[4] = FLAG;

	return buf;
}

unsigned char *createFrame(unsigned char *buffer, int length) {
	int i;
	unsigned char BCC2 = 0x00;

	FRAME_SIZE = length + 6;
```

```
	unsigned char *frame = (unsigned char *)malloc(length + 6);

	frame[0] = FLAG;
	frame[1] = A;
	frame[2] = dl->ns << 6;
	frame[3] = frame[1] ^ frame[2];

	for (i = 0; i < length; i++) {
		BCC2 ^= buffer[i];
		frame[i + 4] = buffer[i];
	}

	frame[length + 4] = BCC2;
	frame[length + 5] = FLAG;

	return frame;
}

int llopen(int fd, int mode) {
	int res = 0;
	unsigned char x, flag;
	unsigned char answer[COMMAND_SIZE];

	switch (mode) {
	case SEND:
		COMMAND = createCommand(SET);

		(void)signal(SIGALRM, connect);
		alarm(dl->timeout);

		if (write(fd, COMMAND, COMMAND_SIZE) == -1) {
			printf("ERROR: Failed to send SET buffer.\n");
			return -1;
		}

		while (STOP == FALSE) {
			res += read(fd, &x, 1);

			if (res == 1 && x == FLAG)
				flag = x;
			else if (res == 1 && x != FLAG)
				res = 0;
			if (x == flag && res > 1)
				STOP = TRUE;

			answer[res - 1] = x;
		}

		if (answer[3] != (A ^ C_UA)) {
			printf("ERROR: Failure on initial connection.\n");
			return -1;
		}

		alarm(0);
		break;
	case RECEIVE:
		while (STOP == FALSE) {
			res += read(fd, &x, 1);
```

```
                if (res == 1 && x == FLAG)
                        flag = x;
                else if (res == 1 && x != FLAG)
                        res = 0;
                if (x == flag && res > 1)
                        STOP = TRUE;

                answer[res - 1] = x;
        }

        if (answer[3] != (A ^ C_SET)) {
                printf("ERROR: Failure on initial connection.\n");
                return -1;
        }

        COMMAND = createCommand(UA);
        if (write(fd, COMMAND, COMMAND_SIZE) == -1) {
                printf("ERROR: Failed to send UA buffer.\n");
                return -1;
        }
        break;
    }

    return 0;
}

int llwrite(int fd, unsigned char *buffer, int length) {
    int res = 0, newSize = 0;
    unsigned char x, flag;
    unsigned char answer[COMMAND_SIZE];

    unsigned char *frame = createFrame(buffer, length);
    STUFFED = stuff(frame, FRAME_SIZE, &newSize);
    STUFFED_SIZE = newSize;

    (void)signal(SIGALRM, send);
    alarm(dl->timeout);

    write(fd, STUFFED, STUFFED_SIZE);

    STOP = FALSE;
    while (STOP == FALSE) {
        res += read(fd, &x, 1);

        if (res == 1 && x == FLAG)
                flag = x;
        else if (res == 1 && x != FLAG)
                res = 0;
        if (x == flag && res > 1)
                STOP = TRUE;

        answer[res - 1] = x;
    }

    if (answer[3] == (A ^ C_RR)) {
        alarm(0);
        triesSend = 0;
        free(STUFFED);
        return STUFFED_SIZE;
    } else if (answer[3] == (A ^ C_REJ)) {
```

```
            alarm(0);
            triesSend = 0;
            stats->rejReceived++;
            return -2;
      }

}

int llread(int fd, unsigned char **buffer) {
      int size = 0;
      volatile int over = FALSE, state = START;
      unsigned char *fileBuf = (unsigned char *)malloc((MAX_SIZE + 10) *
2);

      while (over == FALSE) {
            unsigned char x;

            if (state != DONE) {
                  if (read(app->fd, &x, 1) == -1)
                        return -1;
            }

            switch (state) {
            case START:
                  if (x == FLAG) {
                        fileBuf[size++] = x;
                        state = FLAG_RCV;
                  }
                  break;
            case FLAG_RCV:
                  if (x == A) {
                        fileBuf[size++] = x;
                        state = A_RCV;
                  } else if (x != FLAG) {
                        size = 0;
                        state = START;
                  }
                  break;
            case A_RCV:
                  if (x != FLAG) {
                        fileBuf[size++] = x;
                        state = C_RCV;
                  } else if (x == FLAG) {
                        size = 1;
                        state = FLAG_RCV;
                  } else {
                        size = 0;
                        state = START;
                  }
                  break;
            case C_RCV:
                  if (x == (fileBuf[1] ^ fileBuf[2])) {
                        fileBuf[size++] = x;
                        state = BCC_OK;
                  } else if (x == FLAG) {
                        size = 1;
                        state = FLAG_RCV;
                  } else {
                        size = 0;
                        state = START;
                  }
```

```
                  break;
            case BCC_OK:
                  if (x == FLAG) {
                        fileBuf[size++] = x;
                        state = DONE;
                  } else if (x != FLAG) {
                        if (size > newMaxSize) {
                              COMMAND = createCommand(REJ);
                              stats->rejSent++;

                              if (write(fd, COMMAND, COMMAND_SIZE) == -1) {
                                    printf("ERROR: Failed to send REJ
buffer.\n");
                                    return -1;
                              }

                              return -2;
                        }

                        fileBuf[size++] = x;
                  }
                  break;
            case DONE:
                  fileBuf[size] = 0;
                  over = TRUE;
                  break;
            }
      }

      int k;
      unsigned char *destuffed = destuff(fileBuf, size);
      unsigned char BCC2 = 0x00;

      if (destuffed[3] != (destuffed[1] ^ destuffed[2])) {
            COMMAND = createCommand(REJ);
            stats->rejSent++;

            if (write(fd, COMMAND, COMMAND_SIZE) == -1) {
                  printf("ERROR: Failed to send REJ buffer.\n");
                  return -1;
            }

            return -2;
      } else if (destuffed[2] != C_SET && destuffed[2] != C_UA &&
destuffed[2] != C_DISC && destuffed[2] != C_RR && destuffed[2] != C_REJ) {
            int i;

            if (destuffed[4] == CTRL_PKG_START || destuffed[4] ==
CTRL_PKG_END)
                  k = destuffed[6] + 3;
            else if (destuffed[4] == CTRL_PKG_DATA)
                  k = (destuffed[6] * 256) + destuffed[7] + 4;

            for (i = 0; i < k; i++)
                  BCC2 ^= destuffed[i + 4];

            if (BCC2 != destuffed[4 + k]) {
                  COMMAND = createCommand(REJ);
                  stats->rejSent++;
```

```
                if (write(fd, COMMAND, COMMAND_SIZE) == -1) {
                        printf("ERROR: Failed to send REJ buffer.\n");
                        return -1;
                }

                return -2;
        } else {
                COMMAND = createCommand(RR);

                if (write(fd, COMMAND, COMMAND_SIZE) == -1) {
                        printf("ERROR: Failed to send RR buffer.\n");
                        return -1;
                }
        }
    }

    free(fileBuf);
    *buffer = destuffed;

    return k - 4;
}

int llclose(int fd, int mode) {
    int res = 0;
    unsigned char x, flag;
    unsigned char answer[COMMAND_SIZE];

    switch (mode) {
    case SEND:
        COMMAND = createCommand(DISC);
        if (write(fd, COMMAND, COMMAND_SIZE) == -1) {
            printf("ERROR: Failed to send DISC buffer.\n");
            return -1;
        }

        STOP = FALSE;
        while (STOP == FALSE) {
            res += read(fd, &x, 1);

            if (res == 1 && x == FLAG)
                flag = x;
            else if (res == 1 && x != FLAG)
                res = 0;
            if (x == flag && res > 1)
                STOP = TRUE;

            answer[res - 1] = x;
        }

        if (answer[3] != (A ^ C_DISC)) {
            printf("ERROR: Failure on closing connection.\n");
            return -1;
        }

        COMMAND = createCommand(UA);
        if (write(fd, COMMAND, COMMAND_SIZE) == -1) {
            printf("ERROR: Failed to send UA buffer.\n");
            return -1;
        }
        break;
```

```
	case RECEIVE:
		STOP = FALSE;
		while (STOP == FALSE) {
			res += read(fd, &x, 1);

			if (res == 1 && x == FLAG)
				flag = x;
			else if (res == 1 && x != FLAG)
				res = 0;
			if (x == flag && res > 1)
				STOP = TRUE;

			answer[res - 1] = x;
		}

		if (answer[3] != (A ^ C_DISC)) {
			printf("ERROR: Failure on closing connection.\n");
			return -1;
		}

		COMMAND = createCommand(DISC);
		if (write(fd, COMMAND, COMMAND_SIZE) == -1) {
			printf("ERROR: Failed to send DISC buffer.\n");
			return -1;
		}

		res = 0;
		STOP = FALSE;
		while (STOP == FALSE) {
			res += read(fd, &x, 1);

			if (res == 1 && x == FLAG)
				flag = x;
			else if (res == 1 && x != FLAG)
				res = 0;
			if (x == flag && res > 1)
				STOP = TRUE;

			answer[res - 1] = x;
		}

		if (answer[3] != (A ^ C_UA)) {
			printf("ERROR: Failure on closing connection.\n");
			return -1;
		}
		break;
	}

	return 0;
}

int closeSerialPort() {
	if (tcsetattr(app->fd, TCSANOW, &dl->oldtio) == -1) {
		perror("tcsetattr");
		return -1;
	}

	close(app->fd);
	return 0;
}
```

```
#endif
```

ByteStuffing.h

```
#ifndef BYTESTUFFING_H
#define BYTESTUFFING_H

#include <stdlib.h>
#include <string.h>
#include "Utils.h"

unsigned char *stuff(unsigned char *buf, int length, int *newSize) {
	int i, j = 0;
	unsigned char *stuffed = (unsigned char *)malloc(length * 2);

	for(i = 0; i < length; i++) {
		if(buf[i] == FLAG && i != 0 && i != (length - 1)) {
			stuffed[j] = ESCAPE;
			stuffed[j + 1] = FLAG ^ 0x20;
			j = j + 2;
		} else if(buf[i] == ESCAPE) {
			stuffed[j] = ESCAPE;
			stuffed[j + 1] = ESCAPE ^ 0x20;
			j = j + 2;
		} else {
			stuffed[j] = buf[i];
			j++;
		}
	}

	*newSize = j;
	return stuffed;
}

unsigned char *destuff(unsigned char *buf, int length) {
	int i, j = 0;
	unsigned char *destuffed = (unsigned char *)malloc(length);

	for(i = 0; i < length; i++) {
		if(buf[i] == ESCAPE && buf[i + 1] == (FLAG ^ 0x20)) {
			destuffed[j] = FLAG;
			j++;
			i++;
		} else if(buf[i] == ESCAPE && buf[i + 1] == (ESCAPE ^ 0x20)) {
			destuffed[j] = ESCAPE;
			j++;
			i++;
		} else {
			destuffed[j] = buf[i];
			j++;
		}
	}

	return destuffed;
}

#endif
```

Alarm.h

```
#ifndef ALARM_H
#define ALARM_H

#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include "Utils.h"

unsigned char *COMMAND, *STUFFED;
int triesSend = 0, triesConnect = 0;

void connect() {
	if (triesConnect < dl->retries) {
		printf("\nNo response. Tries left = %d\n", dl->retries -
triesConnect);
		write(app->fd, COMMAND, COMMAND_SIZE);
		triesConnect++;
		alarm(dl->timeout);
	} else {
		alarm(0);
		printf("\nERROR: Failed to create a connection.\n");
		exit(-1);
	}
}

void send() {
	if (triesSend < dl->retries) {
		printf("\nNo response. Tries left = %d\n", dl->retries -
triesSend);
		write(app->fd, STUFFED, STUFFED_SIZE);
		triesSend++;
		stats->frameResent++;
		stats->numberTimeout++;
		alarm(dl->timeout);
	} else {
		alarm(0);
		printf("\nERROR: Failed to send frame.\n");
		exit(-1);
	}
}

#endif
```

Utils.h

```
#ifndef UTILS_H
#define UTILS_H

#define TRUE		1
#define FALSE		0

#define SEND		0
```

```
#define RECEIVE          1

#define UA               0
#define SET              1
#define DISC             2
#define RR               3
#define REJ              4

#define A                0x03
#define FLAG             0x7E
#define DATA             0x01
#define ESCAPE           0x7D

#define C_UA             0x03
#define C_SET            0x07
#define C_DISC           0x0B
#define C_RR             0x05
#define C_REJ            0x01

#define START            0
#define FLAG_RCV         1
#define A_RCV            2
#define C_RCV            3
#define BCC_OK           4
#define DONE             5

#define COMMAND_SIZE     5
#define DATA_PKG_SIZE    4
#define CTRL_PKG_SIZE    11

#define CTRL_PKG_DATA    1
#define CTRL_PKG_START   2
#define CTRL_PKG_END     3

#define PARAM_FILE_SIZE  0
#define PARAM_FILE_NAME  1

#define _POSIX_SOURCE    1
#define PROGRESS_BAR     45
#define MAX_SIZE         512
#define BAUDRATE         B38400

#include <stdio.h>
#include <termios.h>
#include <sys/stat.h>

typedef struct {
      int fd, mode;
      char *port, *fileName;
} Application;

typedef struct {
      int ns, retries, timeout;
      struct termios oldtio, newtio;
} DataLink;

typedef struct {
      int numberTimeout;
      int rejSent, rejReceived;
      int frameSent, frameResent, frameReceived;
```

```
} Statistics;

DataLink *dl;
Application *app;
Statistics *stats;

volatile int FRAME_SIZE, STUFFED_SIZE;

long int getFileSize(char *fileName) {
	struct stat st;

	if (stat(fileName, &st) == 0)
		return st.st_size;

	printf("ERROR: Could not get file size.\n");
	return -1;
}

void printBuffer(unsigned char *buf, int size) {
	int i;
	for (i = 0; i < size; i++)
		printf("0x%02x ", buf[i]);
}

void printProgress(float curr, float total) {
	float per = (100.0 * curr) / total;
	printf("Completed: %6.2f%% [", per);

	int i, pos = (per * PROGRESS_BAR) / 100.0;
	for (i = 0; i < PROGRESS_BAR; i++) {
		if (i <= pos)
			printf("=");
		else
			printf(" ");
	}

	printf("]\r");
	fflush(stdout);
}

void printStatistics() {
	printf("\n - Statistics - \n");
	printf(" # Number of timeouts: %d\n", stats->numberTimeout);
	printf(" # Number of REJs sent: %d\n", stats->rejSent);
	printf(" # Number of REJs received: %d\n", stats->rejReceived);
	printf(" # Number of Frames sent: %d\n", stats->frameSent);
	printf(" # Number of Frames resent: %d\n", stats->frameResent);
	printf(" # Number of Frames received: %d\n", stats->frameReceived);
}

#endif
```