Redes de Computadores

Protocolo de Ligação de Dados

Licenciatura em Engenharia Informática e Computação

Henrique Pinho
up201805000@up.pt

Tiago Gonçalves
up201905179@up.pt

2022

Sumário

No âmbito da unidade curricular Redes de Computadores foi-nos proposta a realização de um trabalho que consiste na implementação de um “Protocolo de ligação de dados” por um meio de transmissão, neste caso utilizando as portas série, que seja capaz de transferir ficheiros entre dois computadores.

A realização deste relatório final serve para complemento do trabalho com o objetivo de auxiliar a sua interpretação, clarificando conceitos e explicando a sua organização.

Introdução

O objetivo do trabalho consiste no desenvolvimento de software que corresponda aos critérios pedidos no guião para um protocolo de ligação de dados.
O relatório foi elaborado com o objetivo de auxiliar a interpretação do código estando este dividido nas seguintes partes:

1. **Arquitetura:** Blocos funcionais e interfaces.
2. **Estrutura do código:** APIs, principais estruturas de dados, principais funções e sua relação com a arquitetura.
3. **Casos de usos principais:** Identificação, sequências de chamada de funções.
4. **Protocolo de ligação lógica:** Identificação dos principais aspetos funcionais e descrição da estratégia de implementação destes aspetos.
5. **Protocolo de aplicação:** Identificação dos principais aspetos funcionais e descrição da estratégia de implementação destes aspetos.
6. **Validação:** Descrição dos testes efetuados com apresentação quantificada dos resultados.
7. **Eficiência do protocolo de ligação de dados:** Caracterização estatística da eficiência do protocolo, feita com recurso a medidas sobre o código desenvolvido.
8. **Conclusões:** Síntese da informação apresentada nas secções anteriores e reflexão sobre os objetivos de aprendizagem alcançados.

Arquitetura

O protocolo de ligação de dados está dividido em duas camadas independentes entre si, sendo estas a camada de ligação de dados e a camada de aplicação.

A camada de ligação de dados (link layer) contém a implementação de código que trata de toda a comunicação com a porta série, ou seja, a sua abertura, escrita, leitura e fecho. É também responsável pela criação e processamento de cada trama, tratando da sua delimitação, transparência, proteção e retransmissão em caso de erro.

A camada de aplicação (application layer), utilizando a camada de ligação de dados, trata do envio e receção de pacotes, quer de dados quer de informação. Cada pacote é

estruturado por esta camada, efetuando o tratamento do seu cabeçalho e definindo a sua numeração.

A independência entre camadas é garantida pelos seguintes factos:

- Ao nível da camada de ligação de dados não existe qualquer distinção entre pacotes de controlo e de dados, nem é relevante a numeração dos pacotes de dados.
- A camada de aplicação desconhece a estrutura das tramas e o respectivo mecanismo de delineação, a existência de stuffing, o mecanismo de protecção das tramas, eventuais retransmissões de tramas de informação, etc. Todas estas funções são exclusivamente realizadas na camada de ligação de dados.

Estrutura do código

O código encontra-se dividido em diferentes ficheiros:

main.c
Responsável pelo tratamento da informação passada nos argumentos e pela inicialização do protocolo.

utils.h
Este ficheiro contém todas as macros necessárias ao programa.

application_layer.c/application_layer.h

- **applicationLayer** - Função principal da camada de aplicação, distingue o recetor do transmissor, invocando as suas respectivas funções.
- **sendFile** - Função que envia o ficheiro a ser transmitido.
- **sendControlPacket**
- **sendDataPacket**
- **readFileInformation**
- **receiveFile -** Função que recebe a informação do ficheiro a ser transmitido.
- **readControlPacket -** Função que recebe o pacote de controlo inicial e final
- **processDataPacket -** Função que escreve a informação recebida do transmissor no recetor.

link_layer.c/link_layer.h

- **alarmHandler -** Função que processa o sinal do alarme após este ter sido iniciado.
- **startAlarm -** Função que inicializa o alarme.
- **disableAlarm -** Função que desativa o alarme.
- **ControlByteCheck -** Função que verifica se o byte de controlo está correto.
- **SMresponse -** Máquina de estados que processa as respostas do recetor e do transmissor.
- **readReceiverResponse -** Função que lê a resposta do recetor.
- **readTransmitterResponse -** Função que lê a resposta do transmissor.
- **llopen -** Função que estabelece e testa a ligação da porta série.
- **readCtrlByte -** Função que lê o byte de controlo e verifica se este está correto.

- **llwrite -** Função que efetua o stuffing, criação da trama e envio desta para o recetor. Após a transmissão, aguarda uma resposta do emissor, agindo em conformidade com a mesma;
- **SMInformationFrame -** Máquina de estados que processa as tramas de informação enviadas pelo transmissor.
- **readTransmitterFrame -** Função que lê a trama de informação enviada pelo emissor
- **verifyFrame -** Função que verifica se existem erros na trama recebida através dos bytes de controlo de erros, designados bcc1 e bcc2;
- **llread -** Função que efetua a leitura da trama, destuffing e envia a resposta para o transmissor de acordo com a trama recebida;
- **llclose -** Função que termina a ligação de dados, encerrando a porta série do transmissor/recetor.
- **openSerialPort -** Função que abre a porta série.

Casos de uso principais

Antes da compilação do programa, o utilizador deverá definir a baudrate e o tamanho do pacote de dados no *"main.c"* , após a compilação, o utilizador deverá passar os seguintes argumentos: porta de série a utilizar, identificação de recetor/transmissor e o nome do ficheiro a ser enviado.
Sequência de chamadas de funções por parte do transmissor:

- Abertura da porta série e estabelecimento da ligação de dados através das funções *openSerialPort* e *llopen*.
- Evocação da função *sendFile*.
- Leitura do nome e do tamanho do ficheiro a ser transmitido através da função *readFileInformation*;
- Criação e envio do pacote de controlo START através da função *sendControlPacket*;
- Criação e envio dos pacotes de dados através da função *sendDataPacket*;
- Criação e envio do pacote de controlo END através da função *sendControlPacket*;
- Fecho da ligação de dados através da função *llclose.*

Sequência de chamadas de funções por parte do recetor:

- Abertura da porta série e estabelecimento da ligação de dados através das funções *openSerialPort* e *llopen*.
- Evocações da função *receiveFile*.
- Leitura das tramas uma a uma efetuada pela função *llread*.
- Processamento do pacote de controlo START através da função *readControlPacket*.
- Processamento dos pacotes de dados através da função *processDataPacket*.
- Processamento do pacote de controlo END através da função *readControlPacket*.
- Fecho da ligação de dados e porta série pela função *llclose*.

Protocolo de ligação lógica

As principais funções do protocolo de ligação lógica são: *llopen, llwrite, llread* e *llclose*.

LLopen: Esta função é responsável pelo estabelecimento e teste da ligação através da porta série entre o transmissor e o recetor que são realizados da seguinte forma:

No lado do transmissor é enviada uma trama de controlo SET, indicando ao recetor que será iniciada uma transferência de dados, ficando a espera de uma trama de controlo UA como resposta do recetor. caso esta não seja recebida ao fim de 3 segundos ocorre um TIMEOUT, outra trama de controlo SET é reenviada pelo transmissor e o processo repete-se, caso sejam obtidos 3 TIMEOUT não é possível estabelecer a ligação e o programa termina.

Por outro lado, o recetor recebendo a trama de controlo SET envia uma resposta com a trama de controlo UA que confirma ao transmissor que está pronto para iniciar a transferência de dados.

LLwrite: Esta função é responsável pelo envio de tramas de informação.

É criado o cabeçalho da trama a ser enviada. De seguida, é feito o *stuffing* da mensagem a ser enviada assim como o cálculo do *bcc2*. Após este processo, é ainda efetuado o *stuffing* do *bcc2*. Por fim, a trama de informação é enviada para a porta série, ficando à espera da resposta do recetor. Caso esta resposta seja negativa, NACK, ou ocorra um TIMEOUT, a função reenvia a trama de informação. Caso contrário, a função termina corretamente uma vez que recebeu uma resposta positiva, ACK.

LLread: Esta função é responsável pela receção de tramas de informação.

É efetuada a leitura da trama byte a byte da porta série. De seguida, é feito o *destuffing* do campo de dados da trama de informação recebida. Depois, são analisados os campos de proteção, ou seja, o *bcc1* e *bcc2* para ser determinada a resposta a ser enviada para o transmissor. Caso algum erro seja detetado, é enviado um NACK, uma resposta negativa. Caso não sejam detetados quaisquer erros, é enviado um ACK, uma resposta positiva e o conteúdo do campo de dados é guardado num *array* passado como parâmetro da função.

LLclose: Esta função é responsável pelo fecho da ligação entre os dois computadores. À semelhança do *llopen* o fecho é realizado da seguinte forma:

No lado do transmissor é enviado uma trama de controlo DISC, indicando ao recetor que a ligação será terminada, ficando a espera de uma trama de controlo DISC por parte do recetor. Caso não esta não seja recebida ao fim de 3 segundos ocorre um TIMEOUT e é reenviada a trama de controlo e o processo repete-se. Caso sejam obtidos 3 TIMEOUT não é possível fechar a ligação lógica e o programa termina.

Do lado do recetor é recebida uma trama de controlo DISC e enviada uma trama semelhante e a ligação é terminada.

Protocolo de Aplicação

As principais funções do protocolo de aplicação são: *sendFile* e *receiveFile*, correspondentes ao transmissor e ao recetor respetivamente.

sendFile: Esta função executa todos os procedimentos necessários à transmissão do ficheiro. Com o auxílio da função *llwrite* da ligação lógica.

É guardada informação relativa ao ficheiro como o seu tamanho e nome com a função *readFileInformation*. De seguida é enviado um pacote de controlo START que contém o tamanho e o nome do ficheiro através da função *sendControlPacket*. Após o pacote de controlo são enviados os pacotes de dados que contém o conteúdo do ficheiro a ser transmitido através da função *sendDataPacket*. Quando todos os pacotes de dados sao enviados, é enviado o pacote de controlo END através da função *sendControlPacket*, que também contém o nome e tamanho do ficheiro.

receiveFile: Esta função efetua a leitura dos pacotes enviados pelo transmissor através da função llread.

Ao receber o pacote de controlo START é criado um ficheiro com o nome contido nesse pacote de controlo, utilizando para isso a função *readControlPacket*. De seguida, são recebidos os pacotes de dados, processados e a informação relativa ao ficheiro é escrita no novo ficheiro através da função *processDataPacket*. No final dos pacotes de dados recebemos o pacote de controlo END com a mesma informação do nome e tamanho do ficheiro inicial, a função *readControlPacket* irá verificar se o tamanho e nome do ficheiro inicial são iguais às do ficheiro final, terminando assim a receção do ficheiro transmitido.

Validação

Para testar o funcionamento correto do trabalho final, foram realizados os seguintes testes:

- Envio de ficheiro sem qualquer interrupção na ligação.
- Envio de um mesmo ficheiro com pacotes de tamanhos diferentes.
- Envio de um mesmo ficheiro com várias *baudrates*.
- Interrupção da ligação da porta série durante o envio do ficheiro.
- Introdução de ruído na porta série durante o envio do ficheiro.

Eficiência do protocolo de ligação de dados

Conclusões

Este projeto teve como objetivo a implementação de um protocolo de ligação de dados para transferência de ficheiros entre dois computadores utilizando portas série como meio de comunicação.Todos os tópicos esperados e referidos no guião foram cumpridos logo, quando o trabalho foi submetido a vários testes obteve sucesso.

Com a realização deste trabalho obtivemos conhecimentos teórico práticos em relação ao tema abordado.

ANEXOS

Main.c

```
// Main file of the serial port project.
// NOTE: This file must not be changed.

#include "./include/application_layer.h"

#define BAUDRATE 9600
#define PACKETSIZE 1024
#define N_TRIES 3
#define TIMEOUT 4
\
// Arguments:
//   $1: /dev/ttySxx
//   $2: tx | rx
//   $3: filename
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, PACKETSIZE);
      return 0;
```

```
}
```

linklayer.c

```
// Link layer protocol implementation
#include "../include/link_layer.h"

// MISC
#define _POSIX_SOURCE 1 // POSIX compliant source
struct termios oldtio;

int alarmFlag;
int alarmCount = 0;

// Alarm function handler
void alarmHandler(int signal)
{
      alarmFlag = 1;
      alarmCount++;
      printf("-Time out: %d.\n", alarmCount);
}
void startAlarm(){

      struct sigaction sa;
    sa.sa_handler = &alarmHandler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
      alarmFlag=0;
    sigaction(SIGALRM, &sa, NULL);

      alarm(3);


}
void disableAlarm(){
      struct sigaction sa;
      sa.sa_handler = NULL;

      sigaction(SIGALRM, &sa, NULL);
      alarmFlag=0;
  alarm(0);
}


////////////////////////////////////////////////
// LLOPEN
////////////////////////////////////////////////
```

```
int ControlByteCheck(unsigned char b){
	return b==CONTROL_BYTE_SET || b==CONTROL_BYTE_UA ||
b==CONTROL_BYTE_DISC
	|| b== CONTROL_BYTE_RR0 || b == CONTROL_BYTE_RR1 || b ==
CONTROL_BYTE_REJ0 || b == CONTROL_BYTE_REJ1;
}
void SMresponse(enum state *currState, unsigned char b, unsigned char*
controlb){
	switch (*currState)
	{
	case START:

	if(b==FLAG)
		*currState=FLG_RCV;
	break;

	case FLG_RCV:
	if(b==ADDRESS_FIELD)
		*currState=A_RCV;
	else if(b!=FLAG)
		*currState=START;
	break;

	case A_RCV:
	if(ControlByteCheck(b)){
		*currState=C_RCV;
		*controlb=b;
	}
	else if(b==FLAG){
		*currState=FLG_RCV;
	}
	else{
		*currState=START;
	}
	break;

	case C_RCV:
	if(b== (ADDRESS_FIELD ^(*controlb))){
		*currState=BCC_OK;
	}
	else if(b==FLAG){
		*currState=FLG_RCV;
	}
	else{
```

```
            *currState=START;
        }
        break;
        case BCC_OK:
        if(b==FLAG){
            *currState=STOP;
        }
        else{
            *currState=START;
        }
        break;
        case STOP:
        break;
        default:
        break;
        }
}

void readReceiverResponse(int fd){
        unsigned char b, controlb;
        enum state state=START;
        while(state!=STOP && alarmFlag==0){
        if (read(fd,&b, 1)<0) {
            printf("-Error reading receiver response.\n\n");
        }
        SMresponse(&state, b, &controlb);
        }

}

void readTransmitterResponse(int fd){

        unsigned char b, controlb;
        enum state state=START;

        while(state!=STOP){
        if (read(fd,&b, 1)<0) {
            printf("-Error reading transmitter response.\n\n");
        }
        SMresponse(&state, b, &controlb);
        }

}

int llopen(int fd, LinkLayer connectionParameters)
{
```

```
printf("\n-----ESTABLISHING CONNECTION-----\n\n");
if (connectionParameters.role==TRANSMITTER) {
unsigned char  ctrlFrame[5];
ctrlFrame[0]=FLAG;
ctrlFrame[1]=ADDRESS_FIELD;
ctrlFrame[2]=CONTROL_BYTE_SET;
ctrlFrame[3]=ctrlFrame[1]^ctrlFrame[2];
ctrlFrame[4]=FLAG;

do{
      write(fd, ctrlFrame, 5);
      printf("-SET Sent.\n");
      alarmFlag= 0;
      startAlarm();
      readReceiverResponse(fd);
      if(!alarmFlag)
            printf("-UA Received.\n");
} while (alarmCount<MAX_TRIES && alarmFlag);
disableAlarm();

if(alarmCount>=MAX_TRIES){
      printf("-Max Tries Exceeded.\n");
      printf("--Could not establish connection.\n");
      printf("---------------------------------\n");
      return -1;
}

} else if (connectionParameters.role==RECEIVER) {
readTransmitterResponse(fd);
printf("-SET received.\n");

unsigned char  ctrlFrame[5];
ctrlFrame[0]=FLAG;
ctrlFrame[1]=ADDRESS_FIELD;
ctrlFrame[2]=CONTROL_BYTE_UA;
ctrlFrame[3]=ctrlFrame[1]^ctrlFrame[2];
ctrlFrame[4]=FLAG;
write(fd, ctrlFrame, 5);
printf("-UA Sent.\n");

} else {
printf("-Unknown flag.\n");
printf("--Could not establish connection.\n");
printf("---------------------------------\n");
return -1;
}
```

```
      printf("\n-----CONNECTION ESTABLISHED-----\n\n");
      return 0;
}

////////////////////////////////////////////////
// LLWRITE
////////////////////////////////////////////////
int ns=0;

int readCtrlByte(int fd, unsigned char *CtrlB){
  unsigned char b;
  enum state state= START;

  while(state!=STOP && alarmFlag==0){
      if(read(fd,&b,1)<0){
      perror("Error reading byte of the receiver response");
      }
      SMresponse(&state,b,CtrlB);
  }

  if(*CtrlB==CONTROL_BYTE_RR0 && ns==1){
      printf("Received postive ACK 0\n");
      return 0;
  }
  else if(*CtrlB==CONTROL_BYTE_RR1 && ns==0){
      printf("Received postive ACK 1\n");
      return 0;
  }
  else if(*CtrlB==CONTROL_BYTE_REJ0 && ns==1){
      printf("Received negative ACK 0\n");
      return -1;
  }
  else if(*CtrlB==CONTROL_BYTE_REJ1 && ns==0){
      printf("Received negative ACK 1\n");
      return -1;
  }
  else {
      return -1;
  }
  return 0;

}

int llwrite(int fd, const unsigned char *buf, int bufSize)
{
      int unsigned nChars = 0;
```

```
alarmCount = 0;

do{
unsigned char frame[2*bufSize+7];

frame[0]=FLAG;
frame[1]=ADDRESS_FIELD;

if(ns==0){
      frame[2]=CONTROL_BYTE_0;
}
else if(ns==1){
      frame[2]=CONTROL_BYTE_1;
}

frame[3]= frame[1] ^ frame[2];

int fIndex=4;

unsigned char bcc2 = 0x00;
for(size_t i=0; i< bufSize;i++){
      bcc2^=buf[i];

      if(buf[i]==FLAG || buf[i]==ESC_BYTE){
            frame[fIndex++]=ESC_BYTE;

            frame[fIndex++]=buf[i]^STUFFING_BYTE;

      }
      else{
            frame[fIndex++]=buf[i];

      }
}

if(bcc2==FLAG || bcc2 == ESC_BYTE){
      frame[fIndex++]=ESC_BYTE;
      frame[fIndex++]=bcc2^STUFFING_BYTE;

}
else{
      frame[fIndex++]=bcc2;

}
frame[fIndex]=FLAG;
```

```
      nChars = write(fd,frame,fIndex+1);
      printf("Sent frame with sequence number %d\n\n",ns);
      printf("wrote %d bytes\n",nChars);

      startAlarm();
      unsigned char CtrlByte;
      if(readCtrlByte(fd, &CtrlByte)==-1){
      disableAlarm();
      alarmFlag=1;
      }

      } while (alarmCount < MAX_TRIES && alarmFlag);
      disableAlarm();

      if(ns==0){
      ns=1;
      }
      else if(ns==1){
      ns=0;
      }

      if(alarmCount>=MAX_TRIES){
      printf("Max tries exceeded\n");
      return -1;
      }

   return nChars;
}

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

void SMInformationFrame(enum state* currentState, unsigned char byte,
unsigned char* controlByte) {

      switch(*currentState){
      case START:

            if(byte == FLAG)
                  *currentState = FLG_RCV;
            break;
      case FLG_RCV:

            if(byte == ADDRESS_FIELD)
                  *currentState = A_RCV;
```

```
            else if(byte != FLAG)
                  *currentState = START;
            break;
      case A_RCV:

            if(byte == CONTROL_BYTE_0 || byte == CONTROL_BYTE_1){
            *currentState = C_RCV;
            *controlByte = byte;
            }
            else if(FLAG == 0x7E){
                  *currentState = FLG_RCV;
            }
            else{
                  *currentState = START;
            }
            break;
      case C_RCV:

            if(byte == (ADDRESS_FIELD^(*controlByte)))
            *currentState = BCC_OK;
            else if(byte == FLAG)
            *currentState = FLG_RCV;
            else
            *currentState = START;

            break;
      case BCC_OK:
            if(byte != FLAG)
            *currentState = DATA_RCV;
            break;
      case DATA_RCV:
            if(byte == FLAG)
            *currentState = STOP;
            break;
      case STOP:
            break;
      }
}

int readTransmitterFrame(int fd, unsigned char* buffer) {

      int pos=0;
      unsigned char byte;
      unsigned char controlByte;
      enum state state = START;
```

```
	while (state != STOP) {

	if(read(fd,&byte,1)<0) {
		printf("erro no byte 'readtransmitterframe'\n");
	}
	SMInformationFrame(&state,byte,&controlByte);
	buffer[pos] = byte;
	pos++;
	}

	return pos;
}


int verifyFrame(unsigned char* frame, int length) {
	unsigned char addressField = frame[1];
	unsigned char controlByte = frame[2];
	unsigned char bcc1 = frame[3];
	unsigned char bcc2 = frame[length-2];
	unsigned char aux = 0x00;
	//verify if bcc1 is correct
	if (controlByte != CONTROL_BYTE_0 && controlByte !=
CONTROL_BYTE_1) {
	printf("Error in control byte\n");
	return -1;
	} else if(bcc1 == (addressField^controlByte)){
	//verify if bcc2 is correct
	for (size_t i = 4; i < length-2; i++) {
		aux^=frame[i];
	}

	if(bcc2 != aux) {
		printf("Error in the data! (bcc2)\n");
		return -2;
	}
	}

	return 0;

}

int llread(int fd,unsigned char *packet)
{

	int received = 0;
	int length = 0;
```

```
	unsigned char controlByte;
	unsigned char auxBuffer[131087];
	int packetIndex = 0;
	alarmCount = 0;

	while (received == 0) {
	length = readTransmitterFrame(fd,auxBuffer);

	printf("frame received\n");

	if (length > 0) {
		unsigned char originalFrame[2*length+7];

		for(size_t i=0; i<4; i++){
			originalFrame[i]=auxBuffer[i];
		}

		int originalFrameIndex = 4;
		int escapeByteFound = FALSE;

		for (int i = 4; i < length-1; i++) {
			if (auxBuffer[i] == ESC_BYTE) {
			escapeByteFound = TRUE;
			continue;
			} else if (auxBuffer[i] == (FLAG^STUFFING_BYTE) &&
escapeByteFound) {
			originalFrame[originalFrameIndex++] = FLAG;
			escapeByteFound = FALSE;
			} else if (auxBuffer[i] == (ESC_BYTE^STUFFING_BYTE) &&
escapeByteFound) {
			originalFrame[originalFrameIndex++] = ESC_BYTE;
			escapeByteFound = FALSE;
			} else {
			originalFrame[originalFrameIndex++] = auxBuffer[i];
			}
		}

		originalFrame[originalFrameIndex] = auxBuffer[length-1];
		controlByte = originalFrame[2];

		if (verifyFrame(originalFrame,originalFrameIndex+1) != 0) {
			if (controlByte == CONTROL_BYTE_0) {
			printf("Rejected frame with 1 as sequence number\n");
			unsigned char frame[5];
			frame[0] = FLAG;
			frame[1] = ADDRESS_FIELD;
```

```
                    frame[2] = CONTROL_BYTE_REJ0;
                    frame[3] = frame[1] ^ frame[2];
                    frame[4] = FLAG;
                    int res = write(fd,frame,5);
                    printf("Sent negative ACK 0 (REJ0) - %d bytes
written\n",res);
                    } else if (controlByte == CONTROL_BYTE_1) {
                    printf("Rejected frame with 0 as sequence number\n");
                    unsigned char frame[5];
                    frame[0] = FLAG;
                    frame[1] = ADDRESS_FIELD;
                    frame[2] = CONTROL_BYTE_REJ1;
                    frame[3] = frame[1] ^ frame[2];
                    frame[4] = FLAG;
                    int res = write(fd,frame,5);
                    printf("Sent negative ACK 1 (REJ1) - %d bytes
written\n",res);
                    }

                    return 0;

              } else {
                    for (size_t i = 4; i < originalFrameIndex-1; i++) {

                    packet[packetIndex++] = originalFrame[i];
                    }

                    if (controlByte == CONTROL_BYTE_0) {
                    unsigned char frame[5];
                    frame[0] = FLAG;
                    frame[1] = ADDRESS_FIELD;
                    frame[2] = CONTROL_BYTE_RR1;
                    frame[3] = frame[1] ^ frame[2];
                    frame[4] = FLAG;
                    int res = write(fd, frame, 5);
                    printf("Sent positive ACK 1 (RR1) - %d bytes
written\n",res);
                    } else if (controlByte == CONTROL_BYTE_1) {
                    unsigned char frame[5];
                    frame[0] = FLAG;
                    frame[1] = ADDRESS_FIELD;
                    frame[2] = CONTROL_BYTE_RR0;
                    frame[3] = frame[1] ^ frame[2];
                    frame[4] = FLAG;
                    int res = write(fd, frame, 5);
                    printf("Sent positive ACK 0 (RR0) - %d bytes
```

```
written\n",res);
                  }
                  received = 1;
            }

      }
      }


      return packetIndex;
}

////////////////////////////////////////////////
// LLCLOSE
////////////////////////////////////////////////
int llclose(int fd, LinkLayer ll, int showStatistics)
{
      printf("\n-----CLOSING CONNECTION-----\n\n");

      if (ll.role == TRANSMITTER) {
      if(alarmCount >= MAX_TRIES){
            return -1;
      }
      unsigned char  ctrlFrame[5];
      ctrlFrame[0]=FLAG;
      ctrlFrame[1]=ADDRESS_FIELD;
      ctrlFrame[2]=CONTROL_BYTE_DISC;
      ctrlFrame[3]=ctrlFrame[1]^ctrlFrame[2];
      ctrlFrame[4]=FLAG;

      do{
            int res = write(fd, ctrlFrame, 5);
            printf("-DISC Sent - %d bytes written.\n",res);
            alarmFlag = 0;
            startAlarm();
            readReceiverResponse(fd);
      } while (alarmCount<MAX_TRIES && alarmFlag);

      if (alarmFlag==0) {
            printf("-DISC received.\n");
      }

      disableAlarm();

      if (alarmCount>=MAX_TRIES) {
            printf("-Max tries exceeded.\n");
```

```
        printf("--Could not close connection.\n");
        printf("-----------------------------\n");
        return -1;
    } else {
        unsigned char  ctrlFrame[5];
        ctrlFrame[0]=FLAG;
        ctrlFrame[1]=ADDRESS_FIELD;
        ctrlFrame[2]=CONTROL_BYTE_UA;
        ctrlFrame[3]=ctrlFrame[1]^ctrlFrame[2];
        ctrlFrame[4]=FLAG;

        int res = write(fd, ctrlFrame, 5);
        printf("-Last UA Sent - %d bytes written\n", res);
        sleep(1);
    }

    } else if (ll.role == RECEIVER) {

    readTransmitterResponse(fd);
    printf("-DISC received.\n");


    unsigned char  ctrlFrame[5];
    ctrlFrame[0]=FLAG;
    ctrlFrame[1]=ADDRESS_FIELD;
    ctrlFrame[2]=CONTROL_BYTE_DISC;
    ctrlFrame[3]=ctrlFrame[1]^ctrlFrame[2];
    ctrlFrame[4]=FLAG;
    int res = write(fd, ctrlFrame, 5);
    printf("-DISC Sent - %d bytes written.\n", res);
    }


    if (showStatistics == TRUE) {
    printf("Statistics\n");
    }

    tcflush(fd, TCIOFLUSH);

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

    close(fd);
```

```
      printf("\n-----CONNECTION CLOSED-----\n\n");
      return 0;
}




int openSerialPort(const char* port, int baudRate)
{
  struct termios newtio;
  int fd;
      printf("BAUD: %d\n", baudRate);
   fd=open(port,O_RDWR | O_NOCTTY);
      if (fd <0) {perror(port); exit(-1);}

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

      bzero(&newtio, 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]      = 10;   /* inter-character timer unused */
      newtio.c_cc[VMIN]       = 1;   /* blocking read until 5 chars
received */

      tcflush(fd, TCIOFLUSH);

      if ( tcsetattr(fd,TCSANOW,&newtio) == -1) {
      perror("tcsetattr");
      exit(-1);
      }
      printf("New termios structure set with baudRate: in:%d |
out:%d\n", newtio.c_ispeed, newtio.c_ospeed);

      return fd;
}
```

application_layer.c

```c
// Application layer protocol implementation

#include "../include/application_layer.h"


LinkLayerRole llrole;
LinkLayer ll;
ControlPacketInformation packetInfo;


void createLinkLayer(int fd,const char* serialPort, LinkLayerRole role,
int baudRate, int nRetransmissions, int timeout, int packetSize) {

      strcpy(ll.serialPort,serialPort);
      ll.fdPort=fd;
      ll.role = role;
      ll.baudRate = baudRate;
      ll.nRetransmissions = nRetransmissions;
      ll.timeout = timeout;
      ll.packetSize=packetSize;

}

int readFileInformation(const char* fileName){

      int fd;
      struct stat status;

      if((fd = open(fileName,O_RDONLY)) < 0){
      perror("Error opening file!\n");
      return -1;
      }

      if(stat(fileName,&status) < 0){
      perror("Error reading file information!\n");
      return -1;
      }

      packetInfo.fdFile=fd;
      packetInfo.fileName = fileName;
      packetInfo.fileSize = status.st_size;

      return 0;
```

```
}


int sendControlPacket(unsigned char controlByte){
      unsigned char packet[5 + strlen(packetInfo.fileName)+
sizeof(packetInfo.fileSize)];
      int pIndex=0;

      packet[pIndex++]=controlByte;

      packet[pIndex++]=FILE_NAME_BYTE;

      packet[pIndex++]= strlen(packetInfo.fileName);


      for(size_t i = 0; i< strlen(packetInfo.fileName);i++){
      packet[pIndex++]= packetInfo.fileName[i];

      }

      packet[pIndex++]=FILE_SIZE_BYTE;

      packet[pIndex++]=sizeof(packetInfo.fileSize);


      packet[pIndex++]= (packetInfo.fileSize >>24 ) & BYTE_MASK;

      packet[pIndex++]= (packetInfo.fileSize >>16 ) & BYTE_MASK;

      packet[pIndex++]= (packetInfo.fileSize >>8 ) & BYTE_MASK;

      packet[pIndex++]= (packetInfo.fileSize) & BYTE_MASK;


      if(llwrite(ll.fdPort,packet, pIndex)<pIndex){
      printf("error writing control packet\n");
      return -1;
      }
      return 0;
}
int sendDataPacket(){

      int numPacketsSent = 0;
      int numPacketsToSend = packetInfo.fileSize/ll.packetSize;
      unsigned char buffer[ll.packetSize];
      int bytesRead = 0;
```

```
	int length = 0;

	if(packetInfo.fileSize%ll.packetSize != 0){
	numPacketsToSend++;
	}
	while(numPacketsSent < numPacketsToSend){

	if((bytesRead = read(packetInfo.fdFile,buffer,ll.packetSize)) <
0){
		printf("Error reading file\n");
	}
	unsigned char packet[4+ll.packetSize];
	packet[0] = CONTROL_BYTE_DATA;
	packet[1] = numPacketsSent % 255;
	packet[2] = bytesRead / 256;
	packet[3] = bytesRead % 256;
	memcpy(&packet[4],buffer,bytesRead);
	length = bytesRead + 4;

	if(llwrite(ll.fdPort,packet,length) < length){
		printf("Error writing data packet to serial port!\n");
		return -1;
	}
	numPacketsSent++;
	}

	return 0;
}
int sendFile(const char* filename){
	if(readFileInformation(filename)<0){
	printf("Could not read file information\n");
	return -1;
	}
	if(sendControlPacket(CONTROL_BYTE_START)<0){
	printf("Could not send start control packet\n");
	return -1;
	}  //start flag
	if(sendDataPacket()<0){
		printf("Could not send data packet\n");
	return -1;
	}//send file
	if(sendControlPacket(CONTROL_BYTE_END)<0){
	printf("Could not send end control packet\n");
	return -1;
	}     //end flag
	return 0;
```

```
}


void applicationLayer(const char *serialPort, const char *role, int
baudRate,
                      int nTries, int timeout, const char *filename,
int packetSize)
{
      int fd = openSerialPort(serialPort, baudRate);

      if (strcmp(role, "tx")==0) {
      llrole = LlTx;
      printf("Transmitter mode engaged\n");
      } else if (strcmp(role, "rx")==0) {
      llrole = LlRx;
      printf("Receiver mode engaged\n");
      } else {
      printf("ERROR: Unknown role\n");
      exit(EXIT_FAILURE);
      }

      createLinkLayer(fd, serialPort, llrole, baudRate, nTries, timeout,
packetSize);

      llopen(ll.fdPort, ll);


      switch (ll.role) {
      case LlTx:
            if (sendFile(filename) < 0) {
                  printf("\nERROR SENDING FILE\n");
            }
            break;
      case LlRx:

      receiveFile();

            break;
      default:
            break;
      }

      llclose(ll.fdPort,ll,0);

}
```

```
int readControlPacket(unsigned char controlByte, unsigned char* packet){
	int packetIndex = 1;
	int fileNameSize = 0;
	int fileSize = 0;
	char* fileNameReceived;



	if (controlByte == CONTROL_BYTE_START) {

	if (packet[packetIndex] == FILE_NAME_BYTE) {

		packetIndex++;
		fileNameSize = packet[packetIndex];
		fileNameReceived = (char*) malloc(packet[packetIndex]+1);
		packetIndex++;

		for (int i = 0; i < fileNameSize; i++) {
			fileNameReceived[i] = packet[packetIndex++];

			if (i == fileNameSize-1) {
			fileNameReceived[fileNameSize] = '\0';
			}
		}

	}

	if(packet[packetIndex] == FILE_SIZE_BYTE){

		packetIndex+=2;
		fileSize += packet[packetIndex++] << 24;
		fileSize += packet[packetIndex++] << 16;
		fileSize += packet[packetIndex++] << 8;
		fileSize += packet[packetIndex];

	}

	packetInfo.fileName = fileNameReceived;
	packetInfo.fileSize = fileSize;

	packetInfo.fdFile = open(fileNameReceived,O_WRONLY | O_CREAT |
O_APPEND, 0664);



	} else if (controlByte== CONTROL_BYTE_END) {
```

```
	if (packet[packetIndex] == FILE_NAME_BYTE) {

		packetIndex++;
		fileNameSize = packet[packetIndex];
		fileNameReceived = (char*) malloc(packet[packetIndex]+1);
		packetIndex++;

		for (int i = 0; i < fileNameSize; i++) {
			fileNameReceived[i] = packet[packetIndex++];

			if (i == fileNameSize-1) {
			fileNameReceived[fileNameSize] = '\0';
			}
		}

	}

	if(packet[packetIndex] == FILE_SIZE_BYTE){
		packetIndex+=2;
		fileSize += packet[packetIndex++] << 24;
		fileSize += packet[packetIndex++] << 16;
		fileSize += packet[packetIndex++] << 8;
		fileSize += packet[packetIndex];

	}

	if (strcmp(packetInfo.fileName,fileNameReceived) != 0) {
		printf("\n\nStart packet and end packet have different file
names \n");
		printf("\n %s %s \n", packetInfo.fileName,
fileNameReceived);
	}
	if (packetInfo.fileSize != fileSize) {
		printf("\n\nStart packet and end packet have different file
size\n");
	}

	}


	return 0;
}

int processDataPacket(unsigned char* buffer) {
```

```
	int infoSize = 256*buffer[2]+buffer[3];

	write(packetInfo.fdFile,buffer+4,infoSize);
	return 0;
}


int receiveFile() {
	unsigned char buffer[ll.packetSize+4];
	int done = 0;
	int lastSequenceNumber = -1;
	int currentSequenceNumber;

	while (done==0) {

	llread(ll.fdPort, buffer);

	if (buffer[0] == CONTROL_BYTE_START) {
		readControlPacket(CONTROL_BYTE_START, buffer);
	}

	if (buffer[0] == CONTROL_BYTE_DATA) {

		currentSequenceNumber = (int)(buffer[1]);
		if(lastSequenceNumber >= currentSequenceNumber &&
lastSequenceNumber != 254) {continue;}

		processDataPacket(buffer);
		lastSequenceNumber = currentSequenceNumber;
	}

	if (buffer[0] == CONTROL_BYTE_END) {
		readControlPacket(CONTROL_BYTE_END, buffer);
		done = 1;
	}
	}
	close(packetInfo.fdFile);
	return 0;
}
```

utils.h

```
#include <termios.h>
```

```
#define FALSE 0
#define TRUE 1

#define TRANSMITTER 0
#define RECEIVER 1

#define FLAG 0x7E
#define ADDRESS_FIELD 0x03
#define CONTROL_BYTE_0   0x00
#define CONTROL_BYTE_1   0x40
#define CONTROL_BYTE_SET 0x03
#define CONTROL_BYTE_UA 0x07
#define CONTROL_BYTE_DISC 0x0B
#define CONTROL_BYTE_REJ0 0x01
#define CONTROL_BYTE_REJ1 0x81
#define CONTROL_BYTE_RR0 0x05
#define CONTROL_BYTE_RR1 0x85


#define CONTROL_BYTE_START 0x02
#define CONTROL_BYTE_END 0x03
#define CONTROL_BYTE_DATA 0x01


#define FILE_SIZE_BYTE 0x00
#define FILE_NAME_BYTE 0x01
#define BYTE_MASK 0xFF

#define ESC_BYTE 0x7D
#define STUFFING_BYTE 0x20


#define MAX_TRIES 3

enum state{START, FLG_RCV, A_RCV, C_RCV, BCC_OK, STOP, DATA_RCV};
```

link_layer.h

```
// Link layer header.
// NOTE: This file must not be changed.

#ifndef _LINK_LAYER_H_
#define _LINK_LAYER_H_
#include "utils.h"
#include <stdio.h>
```

```c
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>

typedef enum
{
	LlTx,
	LlRx,
} LinkLayerRole;

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

typedef struct
{
	const char* fileName;
	int fileSize;
	int fdFile;

} ControlPacketInformation;

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

// MISC
#define FALSE 0
#define TRUE 1

void alarmHandler(int signal);
void startAlarm();
void disableAlarm();
```

```
int ControlByteCheck(unsigned char b);
void SMresponse(enum state *currState, unsigned char b, unsigned char*
controlb);
void readReceiverResponse(int fd);
void readTransmitterResponse(int fd);
int llopen(int fd, LinkLayer connectionParameters);

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

void SMInformationFrame(enum state* currentState, unsigned char byte,
unsigned char* controlByte);
int readTransmitterFrame(int fd, unsigned char* buffer);
int verifyFrame(unsigned char* frame, int length);
int llread(int fd, unsigned char *packet);

int llclose(int fd, LinkLayer ll,int showStatistics);

int openSerialPort(const char *port, int baudRate);


#endif // _LINK_LAYER_H_
```

application_layer.h

```
// Application layer protocol header.
// NOTE: This file must not be changed.

#ifndef _APPLICATION_LAYER_H_
#define _APPLICATION_LAYER_H_
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include "../include/link_layer.h"


void createLinkLayer(int fd, const char* serialPort, LinkLayerRole role,
int baudRate, int nRetransmissions, int timeout, int packetSize);

int readFileInformation(const char* fileName);
int sendControlPacket(unsigned char controlByte);
int sendDataPacket();
int sendFile(const char* filename);
```

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


int readControlPacket(unsigned char controlByte, unsigned char* packet);
int processDataPacket(unsigned char* buffer);
int receiveFile();

#endif // _APPLICATION_LAYER_H_
```