Computer Networks

First Laboratory Work

Gabriel Carvalho (up202208939@up.pt)
Vasco Melo (up202207564@up.pt)

Summary:

This project, carried out as part of the Computer Networks course unit, aims at implementing a data communication protocol for file transmission using a Serial Port.

Through this project, we were able to challenge ourselves and put our knowledge about the Stop-and-Wait protocol to the test in order to consolidate everything we were taught in the lectures.

Introduction:

The objective of this project was to develop and test a data link protocol for transferring files through a serial port. The goal was to implement a reliable communication mechanism using the Stop-and-Wait protocol, ensuring error-free transmission of data between two devices.

This report is structured into eight sections, each focusing on a specific aspect of the project:

1. **Architecture**: This section provides an overview of the functional blocks and interfaces used in the project.
2. **Code Structure**: Here, the main data structures and functions used in the project code are presented.
3. **Main Use Cases**: This section identifies the key operations of the project and the sequence of function calls that occur during the file transfer process.
4. **Logical Link Protocol**: The working mechanism of the logical link protocol is explained in this section.
5. **Application Protocol**: The working mechanism of the application protocol is explained in this section.
6. **Validation**: This section highlights all the different tests and validation methods we used to verify the reliability of our system.
7. **Efficiency of the Data Link Protocol**: Here we show the results and conclusions, as well as graphs, that the tests described in the previous section made us arrive at.
8. **Conclusions**: The final section synthesizes the information presented in the previous one, some reflections and conclusions as well as potential improvements.

1. Architecture:

Functional Blocks

The **Link Layer**, or data link layer, consists of the implementation of the previously mentioned protocol, found in the files *link_layer.h* and *link_layer.c* with related auxiliary functions in *macros.h*. This layer is responsible for establishing and terminating the connection, creating and sending data frames through the serial port, validating the received frames and sending error messages if any issues occur during the transmission.

The **Application Layer**, implemented in *application_layer.h* and *application_layer.c* with related auxiliary functions in *application_layer_aux.h*, manages the transmission of fragmented file data. The file to be transmitted is divided into smaller fragments, each of which is encapsulated in a data packet. These packets are passed to the LinkLayer one by one for transmission. In addition to data packets, the ApplicationLayer also uses control packets for tasks such as synchronization, flow control, and error handling. The ApplicationLayer is responsible for interpreting and processing these packets, ensuring that the fragmented file data is correctly reassembled. It also handles the sequencing of packets and manages retransmissions in case of packet loss, ensuring a reliable file transfer.

Interfaces

The program is executed using two terminals, one on each computer. One terminal runs in **transmitter** mode, while the other runs it in **receiver** mode. The transmitter is responsible for sending the file data, which is fragmented and encapsulated into packets, through the serial port. On the receiver side, the program listens for incoming packets, reassembles the fragments, and reconstructs the original file.

2. Code Structure:

Application Layer

The **main function** for the application layer can be found here:

```C/C++
// Application layer main function.
void applicationLayer(const char *serialPort, const char *role, int
baudRate, int nTries, int timeout, const char *filename);
// Starts the process of sending a file(start control word, file content and
end control word)
int sendFile(const char *filename);
// Starts the process of receiving a file
int receiveFile(const char *filename);
// Starts the process of sending the content of a file
int sendFileContent(unsigned char* file_content, long int file_size);
// Creates the data frame packet being sent
```

```
unsigned char* create_data_frame_packet(unsigned char* data_frame, int
data_frame_size, int* packet_size, unsigned char sequence_number);
// Parses the data frame packet being received
unsigned char* receive_data_frame_packet(unsigned char* data_frame_packet,
int data_frame_packet_size, int* data_size);
```

Link Layer

The **main data structures** for the link layer can be found here:

```C/C++
typedef enum{
    LlTx,LlRx,
} LinkLayerRole;
typedef struct
{
    char serialPort[50];
    LinkLayerRole role;
    int baudRate;
    int nRetransmissions;
    int timeout;
} LinkLayer;

// state machines:
typedef enum {
    START,FLAG_RCV,A_RCV,C_RCV,STOP_BIG,STOP_SMALL,BCC_OK,DATA,END
} State;
typedef enum{
    SET,UA,RR0,RR1,REJ0,REJ1,DISC,FRAME1,FRAME0,ERROR,NOTHING_C
} C_TYPE;
```

The **main functions** for link layer can be found here:

```C/C++
// Open a connection using the "port" parameters defined in struct
linkLayer.
int llopen(LinkLayer connectionParameters);
// Send data in buf with size bufSize.
int llwrite(const unsigned char *buf, int bufSize);
// Receive data in packet.
int llread(unsigned char *packet);
// Close previously opened connection.
int llclose(int showStatistics);
// Encodes the data being sent
unsigned char* stuffing_encode(const unsigned char *buf, int bufSize, int
*newBufSize);
```

```
// Decodes the data being received
unsigned char* stuffing_decode(unsigned char *buf, int bufSize, int
*newBufSize);
// Calculates the BCC2
unsigned char calculate_BCC2(const unsigned char *buf, int bufSize);
// Handler function for the data received in llwrite
int handle_llwrite_reception();
// Handler function for the data received in llread
int handle_llread_reception(unsigned char *buf, int bufSize);
```

3. Main Use Cases:

The program can be executed as either transmitter or receiver, and depending on which is chosen, the main functions being executed will differ.

Transmitter

Call sequence for the transmitter:

1. **llopen():** establishes a connection between transmitter and receiver
2. **sendFile():** starts the process of sending a file
3. **create_data_frame_packet():** creates the data frame packet
4. **llwrite():** creates and sends, through the serial port, an information frame, based on the packet received as argument
5. **stuffing_encode():** encodes the data being written
6. **calculateBBC2():** calculates the BCC2 to place it into the frame
7. **handle_llwrite_reception():** handles the reception sent by the receiver and dictates the next course of action
8. **llclose():** severs the connection between transmitter and receiver

Receiver

Call sequence for the receiver:

1. **llopen():** establishes a connection between transmitter and receiver
2. **receiveFile():** starts the process of receiving a file
3. **llread():** state machine for managing and validation of the control and data frames, sent by the transmitter
4. **stuffing_decode():** decodes the data being read
5. **handle_llread_reception():** handles the reception sent by the transmitter and dictates the next course of action
6. **receive_data_frame_packet():** parses the data frame packet received
7. **llclose():** severs the connection between transmitter and receiver

4. Logical Link Protocol

The data link layer enables reliable serial communication between the transmitter and receiver. This implementation uses the **Stop-and-Wait** protocol to manage connection establishment, data transfer, and termination.

The **llopen** function establishes the connection. The transmitter sends a SET frame and waits for a UA (Unnumbered Acknowledgment) frame from the receiver. A timeout mechanism is implemented to handle retries if no response is received.

Once the connection is established, the **llwrite** function handles data transmission. The data is byte-stuffed to prevent conflicts with control characters, then framed and sent.

The **llread** function handles reception, validating frames via BCC1 and BCC2 checks. Data is de-stuffed, and if valid, it is passed to the higher layer.

The **llclose** function handles connection termination. The transmitter sends a DISC frame and waits for an acknowledgment. After receiving the DISC, it sends a UA frame to close the connection.

5. Application Protocol

The application layer is responsible for managing and transferring files between systems. This layer directly interacts with the file being transferred and with the user through the function **applicationLayer**, which accepts the following parameters:

```
C/C++
void applicationLayer(const char *serialPort, const char *role, int
baudRate, int nTries, int timeout, const char *filename)
```

These parameters allow configuration of:

- The serial port for communication
- The role of the device (transmitter or receiver)
- The transmission speed (baudRate)
- The maximum number of retransmission attempts (nTries)
- The maximum wait time for a response (timeout)
- The name of the file to be transferred

The file transfer is managed through the **LinkLayer API**, which converts data packets into information frames. The process starts with the connection establishment through the function **llopen**. Once the connection is established, the file content is processed differently depending on the role of the device:

Transmitter:

On the transmitter side, the sending process is managed by the function **sendFile**, which implements a sequence of operations to ensure reliable data transfer. Initially, the file is opened in binary mode, and its total size is determined through file pointer positioning operations. The transmitter then creates a control packet using the function **create_control_frame**, which uses a TLV (Type, Length, Value) structure to encapsulate essential metadata about the file. This packet includes not only the file name but also its total size, allowing the receiver to properly allocate the resources needed for reception. After the successful transmission of this initial control packet, the transmitter proceeds to read the entire file content into a memory buffer, preparing it for fragmentation and transmission.

The next phase involves the fragmentation and transmission of the file content, managed by the function **sendFileContent**. This function implements a sliding window mechanism, where the file content is divided into smaller fragments, each encapsulated in a data packet via the function **create_data_frame_packet**. Each data packet is carefully structured with a control field identifying it, a sequence number to ensure correct order during reception, and size fields specifying the amount of data contained in the packet. The process continues iteratively until all the file content has been transmitted, with the transmitter tracking progress through the byte counter.

Receiver:

On the receiver side, the process is managed by the function **receiveFile**, which ensures correct and ordered reception of data. The receiver begins by waiting for the initial control packet, which is processed by the function **receiveControlPacket**. This function extracts crucial information about the file to be received, including its total size and name, allowing the receiver to allocate the necessary resources and prepare the output file.

After the initial phase, the receiver enters a data reception loop, where each packet received through the function **llread** is processed by **receive_data_frame_packet**. This function performs several checks:

First, it confirms whether the received packet is indeed a data packet by verifying its control field. Then, it extracts and validates the size fields to determine the amount of data contained in the packet. The receiver carefully tracks the sequence of received packets, ensuring that no data is lost or duplicated during the transfer. When a packet is successfully received, its content is extracted and written to the output file, thus maintaining data integrity throughout the transfer process.

During all these processes, the system implements error checking at various stages, including validation of the initial connection, packet sequence checking, receipt acknowledgment, and retransmission in case of failure. The transfer is complete when all packets have been successfully transmitted or when the maximum number of retransmission attempts has been reached. The connection is then severed through the function **llclose**.

6. Validation

Throughout the development of the project, several tests were conducted with the aim of evaluating different scenarios in which the Stop-and-Wait protocol might encounter challenges, ensuring that file transfer would still be reliable. The following tests were performed:

- Different file attributes (size and name);
- Different baud rates;
- Different frame sizes;
- Partial and/or complete interruptions of the serial port;
- Introduction of noise on the serial port via short-circuiting;
- Rejection of data frames by introducing randomness.

7. Data Link Protocol Efficiency

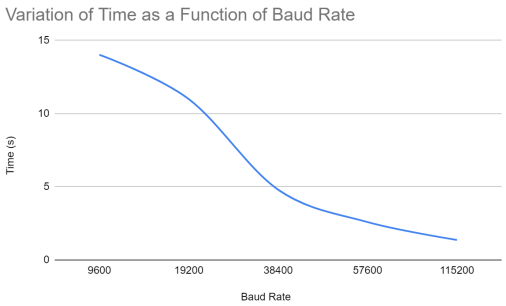
In order to obtain the following data, these respective formulas were used:

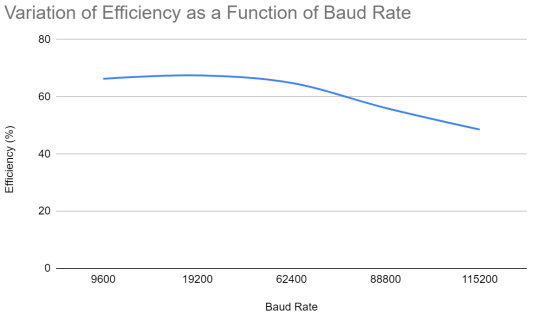
$$FER = 1 - (1 - BER)^n \quad E(\%) = \frac{bit\ rate}{baudrate} * 100$$

$$Practical\ Error\ Percentage\ (\%) = (1 - \frac{good\ frames}{total\ frames\ received}) * 100$$

Variation of the baud rate

Considering a file of 10 968 bytes, a fixed error rate of 0.05 FER (Frame Error Rate), and a frame size of 1000 bytes, the transfer time and protocol efficiency were analyzed as a function of variations in baud rate.

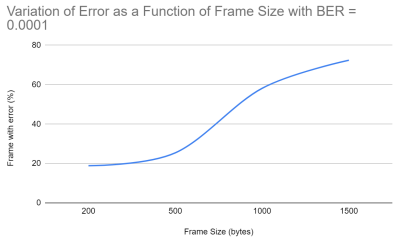


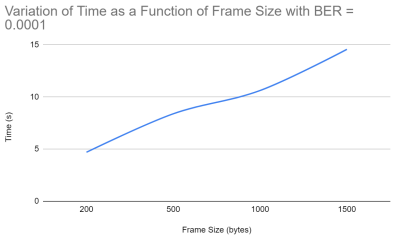


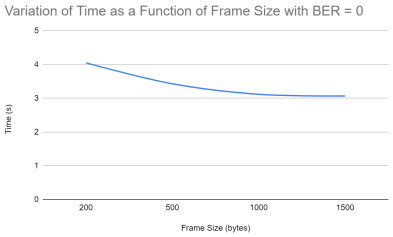
With the increase in baud rate, it was observed that the transfer time decreases proportionally, but the protocol efficiency also gradually decreases. This happens because, despite the increase in the transmission speed, the reduction in the frame error rate and the higher occurrence of retransmissions lead to a decrease in the overall transfer efficiency.

Variation of Frame Size with and without Error Rate

To test the effect of frame size variation, an experiment was conducted with a file of 10 968 bytes and a constant baud rate of 38 400. The impact on transfer time and error was evaluated in two environments: one with low error (BER = 0) and another with a more unstable environment (BER = 0.0001).



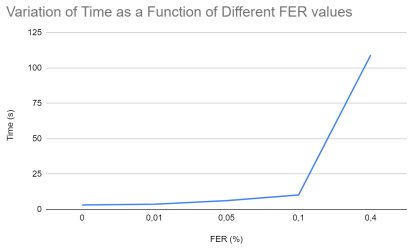




With a BER of 0, it was observed that the protocol's transfer time decreases as the frame size increases. However, when errors are introduced, the transfer time increases with the frame size, as the FER (Frame Error Rate) increases with larger frames, leading to more retransmissions and longer file transfer completion times.

Variation of FER Rate

With a file of 10 968 bytes, a baud rate of 38 400, and a frame size of 1000 bytes, the transfer time, efficiency, and protocol error were analyzed for different FER (Frame Error Rate) values.



As the FER increases, it can be observed that the transfer time grows exponentially, due to the higher number of frame retransmissions required to compensate for errors. This results in a significant delay in file transfer completion.

8. Conclusions

Based on the tests conducted and the results obtained, it can be concluded that the Stop-and-Wait protocol is highly reliable, as the file transmission is not compromised under various conditions. However, the efficiency and transfer time of a file can be easily impacted by several factors, notably an increase in FER and the need for frame acknowledgment before sending the next one. To mitigate the impact of errors, a potential solution would be to adjust the frame size based on the receiver's feedback.

For instance, if a REJ (Reject) is received, the frame size could be reduced until successful transmission is possible, and once the frame is accepted, the size of subsequent frames would increase. To further enhance file transfer efficiency, other protocols, such as Go-Back-N or Selective Repeat, could be employed, as they allow the transmission of multiple frames without waiting for individual acknowledgments for each frame.

Appendix I - Source Code

macros.h

```C/C++
#ifndef MACROS_H
#define MACROS_H

#include <stdbool.h>

typedef enum {
	START,
	FLAG_RCV,
	A_RCV,
	C_RCV,
	STOP_BIG,
	STOP_SMALL,
	BCC_OK,
	DATA,
	END
} State;

typedef enum{
	SET,
	UA,
	RR0,
	RR1,
	REJ0,
	REJ1,
	DISC,
	FRAME1,
	FRAME0,
	ERROR,
	NOTHING_C
} C_TYPE;


//global variables
#define FLAG 0x7E
#define A_Tx 0x03
#define A_Rx 0x01

#define C_SET 0x03
#define C_UA 0x07
#define C_RR0 0xAA
#define C_RR1 0xAB
#define C_REJ0   0x54
```

```
#define C_REJ1   0x55
#define C_DISC   0x0B
#define C_FRAME0 0x00
#define C_FRAME1 0x80

unsigned char set_message[5] = {FLAG,A_Tx,C_SET,A_Tx^C_SET,FLAG};
unsigned char ua_message_tx[5] = {FLAG,A_Tx,C_UA,A_Tx^C_UA,FLAG};
unsigned char ua_message_rx[5] = {FLAG,A_Rx,C_UA,A_Rx^C_UA,FLAG};

unsigned char rr0_message[5] = {FLAG,A_Tx,C_RR0,A_Tx^C_RR0,FLAG};
unsigned char rr1_message[5] = {FLAG,A_Tx,C_RR1,A_Tx^C_RR1,FLAG};

unsigned char rej0_message[5] = {FLAG,A_Tx,C_REJ0,A_Tx^C_REJ0,FLAG};
unsigned char rej1_message[5] = {FLAG,A_Tx,C_REJ1,A_Tx^C_REJ1,FLAG};

unsigned char disc_message[5] = {FLAG,A_Tx,C_DISC,A_Tx^C_DISC,FLAG};
unsigned char disc_message_rx[5] = {FLAG,A_Rx,C_DISC,A_Rx^C_DISC,FLAG};

int llopenRx();
int llopenTx();
unsigned char* read_aux(int *readBytes, bool alarm);
int write_aux(unsigned char *message, int numBytes);
unsigned char* stuffing_encode(const unsigned char *buf, int bufSize, int
*newBufSize);
unsigned char* stuffing_decode(unsigned char *buf, int bufSize, int
*newBufSize);
unsigned char calculate_BCC2(const unsigned char *buf, int bufSize);
int mount_frame_message(int numBytesMessage, unsigned char *buf, unsigned
char *frame);
void final_check();
void debug_write(unsigned char *message, int numBytes);
int c_check(unsigned char byte);
int handle_llwrite_reception();
int handle_llread_reception(unsigned char *buf, int bufSize);


#endif
```

application_layer_aux.h

```
C/C++
#ifndef APPLICATION_LAYER_AUX
#define APPLICATION_LAYER_AUX

#define DataSizeL1(n)   (n >> 8) & 0xFF
```

```
#define DataSizeL2(n)   n & 0xFF
#define DATA_SIZE(n,m)  ((n << 8) | m)

int sendFile(const char *filename);
int receiveFile(const char *filename);
unsigned char* create_control_frame(unsigned char c, const char* filename,
long int file_size, unsigned int* final_control_size);
unsigned char* receiveControlPacket(unsigned char* control_frame, unsigned
long int* file_size, int* filename_size);
void debug_control_frame_rx(unsigned char* filename, unsigned int
filename_size, int size_of_file);
void debug_control_frame_tx(const char* filename, unsigned int
filename_size, int size_of_file);
unsigned char calculate_octets(long int file_size);
int sendFileContent(unsigned char* file_content, long int file_size);
unsigned char* create_data_frame_packet(unsigned char* data_frame, int
data_frame_size, int* packet_size, unsigned char sequence_number);
unsigned char* receive_data_frame_packet(unsigned char* data_frame_packet,
int data_frame_packet_size, int* data_size);
void debug_print_frame(unsigned char* frame, int frame_size);
void display_statistics();

#endif
```

link_layer.c

C/C++

```
// Link layer protocol implementation

#include "link_layer.h"
#include "serial_port.h"
#include <stdlib.h>
#include <stdbool.h>
#include "macros.h"
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/time.h>

// MISC
#define _POSIX_SOURCE 1 // POSIX compliant source

struct timeval start, end;
```

```
//Global Variables

bool debug = false;

State state = END;
C_TYPE control = NOTHING_C;
bool frame_num = 0;
bool my_role;

int alarmCounter = 0;
int timeout = 0;
int restransmissions = 0;
bool alarmFlag = false;

int num_errors = 0;

unsigned char buf[2*MAX_PAYLOAD_SIZE] = {0};
long int bytes_sent = 0;
long int bytes_received = 0;

//active debug


void alarmHandler(int signal){
	alarmFlag = false;
	alarmCounter++;
	printf("Alarmcounter: %d\n", alarmCounter);
}


////////////////////////////////////////////////
// LLOPEN
////////////////////////////////////////////////
int llopen(LinkLayer connectionParameters){
	gettimeofday(&start, NULL);
	if (openSerialPort(connectionParameters.serialPort,
			connectionParameters.baudRate) < 0){
	return -1;
	}

	timeout = connectionParameters.timeout;
	restransmissions = connectionParameters.nRetransmissions;

	if(connectionParameters.role == LlRx){
	// Wait for a SET then ...
	// Send UA
	my_role = 0;
	if(!llopenRx()){
```

```
                perror("Error llopenRx");
                return -1;
        }

        } else {
        // Send SET (write)
        // Wait UA (read)
        my_role = 1;
        if(!llopenTx()){
                perror("Error llopenTx");
                return -1;
        }
        }


        return 1;
}


int llopenRx(){


        control = NOTHING_C;
        unsigned char *readBuf;
        int numBytes;

        while(control != SET){
        readBuf = read_aux(&numBytes, false);

        if(readBuf == NULL){
                perror("Error reading UA");
                free(readBuf);
                return -1;
        }
        if(readBuf != NULL){
                free(readBuf);
        }
        }

        control = NOTHING_C;


        write_aux(ua_message_tx, 5);

        return 1;
}

//return 1 -> UA
```

```
//return 0 -> Tries expired
int llopenTx(){
	(void) signal(SIGALRM, alarmHandler);

	unsigned char *readBuf;
	int numBytes;
	int tries = restransmissions;
	control = NOTHING_C;
	alarmCounter = 0;


	while(alarmCounter < tries && control != UA){
	alarmFlag = true;
	alarm(timeout);
	write_aux(set_message, 5); // Send SET
	readBuf = read_aux(&numBytes, true);

	if(readBuf != NULL){
		if(control == UA){
		break;
		}
		else{
		alarmCounter = 0;
		num_errors++;
		}
	}
	if(readBuf != NULL){
		free(readBuf);
	}
	}




	return control == UA;
}


////////////////////////////////////////////////
// LLWRITE
////////////////////////////////////////////////

//vai montar a frame neste formato: FLAG A C BCC1 D1 D2 ... Dn BCC2 FLAG
// A -> 0x03
// C -> 0x00 ou 0x80 (dependendo do frame)
// calcular BCC2
// byte stuffing
// montar e enviar a frame
```

```
// esperar por RR ou REJ

int llwrite(const unsigned char *buf, int bufSize){
       int localBufSize = 0;

       unsigned char bcc2 = calculate_BCC2(buf, bufSize);
       unsigned char *newBuf;

       unsigned char *buf_bcc2 = (unsigned char *)malloc(bufSize + 1 *
sizeof(unsigned char));
       memcpy(buf_bcc2, buf, bufSize * sizeof(unsigned char));
       buf_bcc2[bufSize] = bcc2;

       bytes_sent += bufSize;

       newBuf = stuffing_encode(buf_bcc2, bufSize + 1, &localBufSize);
       free(buf_bcc2);

       if(newBuf == NULL){
       perror("Error encoding stuffing");
       return -1;
       }

       unsigned char frame[localBufSize + 6];

       int frameSize = mount_frame_message(localBufSize, newBuf, frame);
       free(newBuf);

       //colocar alarme
       bool flag = true;
       int tries = restransmissions;

       int numBytes;
       unsigned char* readBuf;


       alarmCounter = 0;


       printf("Frame number write: %d\n", frame_num);

       while(alarmCounter < tries && flag){
       alarmFlag = true;
       alarm(timeout);

       bytes_received += frameSize;

       if(!write_aux(frame, frameSize)){
```

```
                perror("Error writtin message");
                return -1;
        }

        control = NOTHING_C;
        readBuf = read_aux(&numBytes, true);

        if(readBuf != NULL){
                int result = handle_llwrite_reception();

                //mensagem recebida com sucesso
                if(result == 1){
                flag = false;
                }else{
                num_errors++;
                alarmCounter = 0;
                }
        }
        if(readBuf != NULL){
                free(readBuf);
        }
        }

        control = NOTHING_C;

        if(flag){
        return -1;
        }

        return bufSize;
}

//receber frame
//byte destuffing
//verificar BCC2
//enviar RR ou REJ dependendo da frame recebida
//return -2 -> SET
//return -3 -> DISC


////////////////////////////////////////////////
// LLREAD
////////////////////////////////////////////////
int llread(unsigned char *packet){
        // TODO
        int numBytes;
        unsigned char *readBuf;
        control = NOTHING_C;
```

```
        bool flag = true;
        unsigned char *newBuf;
        int newBufSize;
        int result;

        while(flag){
        readBuf = read_aux(&numBytes, false);

        if (readBuf == NULL){
                perror("Error read_aux function");
                free(readBuf);
                return -1;
        }

        bytes_received += numBytes;

        //destuffing
        newBuf = stuffing_decode(readBuf, numBytes, &newBufSize);

        //verificar se há erros
        result = handle_llread_reception(newBuf, newBufSize);

        if(newBuf != NULL){
                free(readBuf);
        }
        control = NOTHING_C;

        //mensagem recebida com sucesso
        if(result == 1){
                newBuf[newBufSize - 1] = '\0';
                frame_num = !frame_num;

                *packet = (unsigned char*)malloc((newBufSize - 1) *
sizeof(unsigned char));
                memcpy(packet, newBuf, (newBufSize - 1) * sizeof(unsigned
char));
                free(newBuf);

                bytes_sent += newBufSize - 1;

                return newBufSize - 1;
        }



        num_errors++;

        }
```

```
        free(newBuf);
        printf("Error reading frame\n");

        return result;
}

////////////////////////////////////////////////
// LLCLOSE
////////////////////////////////////////////////
int llclose(int showStatistics){

        unsigned char *readBuf;
        int numBytes;
        (void) signal(SIGALRM, alarmHandler);
        int tries = restransmissions;
        bool flag = true;

        //enviar DISC e receber UA, caso contrário reenviar DISC

        if(my_role == 0){
        while(flag){

                control = NOTHING_C;
                readBuf = read_aux(&numBytes, false);

                if(control == DISC){
                //enviar UA
                write_aux(disc_message, 5);
                flag = false;
                }

                if(readBuf != NULL){
                free(readBuf);
                }
        }


        flag = true;


        while(flag){
                //read UA

                control = NOTHING_C;
                readBuf = read_aux(&numBytes, false);

                if(readBuf != NULL){
```

```
        alarmCounter = 0;
        if(control == DISC){
        //enviar DISC
        write_aux(disc_message, 5);
        } else if(control == UA){
                flag = false;
        }
        }
        if(readBuf != NULL){
        free(readBuf);
        }
}
}
//enviar DISC, receber DISC e enviar UA
else{
while (alarmCounter < tries && flag){
        //enviar DISC
        alarmFlag = true;
        write_aux(disc_message, 5);

        //receber DISC
        control = NOTHING_C;
        readBuf = read_aux(&numBytes, true);

        if(readBuf != NULL){
        alarmCounter = 0;
        if(control == DISC){
                //enviar UA
                write_aux(ua_message_tx, 5);
                flag = false;

        }
        }
        if(readBuf != NULL){
        free(readBuf);
        }
}

control = NOTHING_C;

}

if(showStatistics){
display_statistics();
}

int clstat = closeSerialPort();
return clstat;
```

```
}

void stateMachineHandler(unsigned char byte, int *pos){
	switch (state){
	case START:
		if(byte == FLAG){
		state = FLAG_RCV;
		buf[*pos] = byte;
		*pos += 1;
		}
		break;

	case FLAG_RCV:
		if(byte == A_Tx || byte == A_Rx){
		state = A_RCV;
		buf[*pos] = byte;
		*pos += 1;
		} else if(byte != FLAG){
		state = START;
		*pos = 0;
		}
		break;

	case A_RCV:
		if(c_check(byte)){
		state = C_RCV;
		buf[*pos] = byte;
		*pos += 1;
		} else if(byte == FLAG){
		state = FLAG_RCV;
		*pos = 1;
		}
		else{
		state = START;
		*pos = 0;
		}

		break;

	case C_RCV:
		if(byte == (buf[1] ^ buf[2])){
		state = BCC_OK;
		buf[*pos] = byte;
		*pos += 1;
		} else if(byte == FLAG){
		state = FLAG_RCV;
		*pos = 1;
		} else {
```

```
                state = START;
                *pos = 0;
                }
                break;
        case BCC_OK:
                if(byte == FLAG){
                state = STOP_SMALL;
                buf[*pos] = byte;
                *pos += 1;
                } else {
                state = DATA;
                buf[*pos] = byte;
                *pos += 1;
                }

                break;

        case DATA:
                if(byte == FLAG){
                state = STOP_BIG;
                buf[*pos] = byte;
                *pos += 1;
                } else {
                state = DATA;
                buf[*pos] = byte;
                *pos += 1;
                }
                break;
        default:
                break;


        }
}

int c_check(unsigned char byte){
        switch (byte){
        case C_SET:
                control = SET;
                break;

        case C_UA:
                control = UA;
                break;

        case C_RR0:
                control = RR0;
                break;
```

```
        case C_RR1:
                control = RR1;
                break;

        case C_REJ0:
                control = REJ0;
                break;

        case C_REJ1:
                control = REJ1;
                break;

        case C_DISC:
                control = DISC;
                break;

        case C_FRAME0:
                control = FRAME0;
                break;

        case C_FRAME1:
                control = FRAME1;
                break;

        default:
                return 0;
                break;
        }

        return 1;
}

//adicionar alarme
unsigned char* read_aux(int *readBytes, bool alarm){
        state = START;
        unsigned char byte;
        int pos = 0;
        if(alarm){
        while(state != STOP_BIG && state != STOP_SMALL && alarmFlag){
                if(readByteSerialPort(&byte)){
                stateMachineHandler(byte, &pos);
                }
        }

        if(!alarmFlag){
                return NULL;
        }
```

```
        }
        else{

        while(state != STOP_BIG && state != STOP_SMALL){
              if(readByteSerialPort(&byte)){
              stateMachineHandler(byte, &pos);
              }
        }
        }


        unsigned char *mensseBuf = (unsigned char *)malloc(pos *
sizeof(unsigned char));
        memcpy(mensseBuf, buf, pos * sizeof(unsigned char));
        *readBytes = pos;

        final_check();

        state = END;

        return mensseBuf;
}

int write_aux(unsigned char* message, int numBytes){

        if(writeBytesSerialPort(message, numBytes) == -1){
        perror("Error sending SET");
        return -1;
        }

        return 1;
}

//0x7e -> 0x7d 0x5e
//0x7d -> 0x7d 0x5d
unsigned char* stuffing_encode(const unsigned char *buf, int bufSize, int
*newBufSize){

        int maxBufSize = 2 * bufSize;
        unsigned char *newBuf = (unsigned char *)malloc(maxBufSize *
sizeof(unsigned char));

        if(newBuf == NULL){
        perror("Error allocating memory");
        return NULL;
        }
```

```
	int j = 0;

	for(int i = 0; i < bufSize; i++){
	if(buf[i] == 0x7e){
		newBuf[j] = 0x7d;
		j++;
		newBuf[j] = 0x5e;
		j++;
	} else if(buf[i] == 0x7d){
		newBuf[j] = 0x7d;
		j++;
		newBuf[j] = 0x5d;
		j++;
	} else {
		newBuf[j] = buf[i];
		j++;
	}
	}

	*newBufSize = j;
	newBuf = (unsigned char *)realloc(newBuf, j * sizeof(unsigned char));
	if(newBuf == NULL){
	perror("Error reallocating memory");
	return NULL;
	}

	return newBuf;
}

//0x7d 0x5e -> 0x7e
//0x7d 0x5d -> 0x7d
//remover duas flags, a, c e bcc1
unsigned char* stuffing_decode(unsigned char *buf, int bufSize, int
*newBufSize){
	int maxBufSize = bufSize;
	unsigned char *newBuf = (unsigned char *)malloc((maxBufSize - 5) *
sizeof(unsigned char));

	if(newBuf == NULL){
	perror("Error allocating memory");
	return NULL;bool get_frame_num(void);
	}

	int j = 0;

	for(int i = 4; i < bufSize - 1; i++){
	if(buf[i] == 0x7d && buf[i+1] == 0x5e){
		newBuf[j] = 0x7e;
```

```
            j++;
            i++;
        } else if(buf[i] == 0x7d && buf[i+1] == 0x5d){
            newBuf[j] = 0x7d;
            j++;
            i++;
        } else {
            newBuf[j] = buf[i];
            j++;
        }
        }

        *newBufSize = j;
        newBuf = (unsigned char *)realloc(newBuf, j * sizeof(unsigned char));
        if(newBuf == NULL){
        perror("Error reallocating memory");
        return NULL;
        }

        return newBuf;
}

//calculate BCC2 = D1 xor D2 xor ...bool get_frame_num(void); xor Dn
unsigned char calculate_BCC2(const unsigned char *buf, int bufSize){
        unsigned char bcc2 = 0x00;
        for(int i = 0; i < bufSize; i++){
        bcc2 ^= buf[i];
        }

        return bcc2;
}

//verify BCC2 == D1 xor D2 xor ... xor Dn
bool verify_BCC2(unsigned char *buf, int bufSize){
        unsigned char bcc2_calc = calculate_BCC2(buf, bufSize - 1);
        return buf[bufSize-1] == bcc2_calc;
}

int mount_frame_message(int numBytesMessage, unsigned char *buf, unsigned
char *frame){
        frame[0] = FLAG;
        frame[1] = A_Tx;

        if(!frame_num){
        frame[2] = C_FRAME0;
        } else {
        frame[2] = C_FRAME1;
        }
```

```
        frame[3] = A_Tx ^ frame[2];
        for(int i = 0; i < numBytesMessage; i++){
        frame[i+4] = buf[i];
        }

        frame[numBytesMessage + 4] = FLAG;
        return numBytesMessage + 5;
}



int handle_llwrite_reception(){
        //frame_0 recebido com sucesso envia o frame_1 ou o contrário

        if((frame_num == 0 && control == RR1) || (frame_num == 1 && control ==
RR0)){
        frame_num = !frame_num;
        return 1;
        }
        //frame_0 foi recebido, mas com erros ou o contrário
        else if((frame_num == 0 && control == REJ0) || (frame_num == 1 &&
control == REJ1)){
        return 0;
        }

        return 0;
}

//return 1 -> RR
//return -1 -> REJ
//return 0 -> Same
//return -2 -> SET
//return -3 -> DISC
//return -4 -> Unknown

int handle_llread_reception(unsigned char *buf, int bufSize){

        //verifica se é SET ou DISC
        if(control == SET){
        //enviar UA
        write_aux(ua_message_tx, 5);
        return -2;
        }
        else if(control == DISC){
        //enviar DISC
        write_aux(disc_message, 5);
```

```
        return -3;
        }

        //verifica o BBC2
        if(!verify_BCC2(buf, bufSize)){
        //enviar REJ
        if(frame_num){
                //enviar REJ0
                write_aux(rej1_message, 5);
        } else {
                //enviar REJ1
                write_aux(rej0_message, 5);
        }
        return -1;
        }

        //verifica se é frame_0 ou frame_1

        if(control == FRAME0){
        //enviar RR1
        write_aux(rr1_message, 5);
        if(frame_num == 0){
                return 1;
        } else {
                return -4;
        }
        } else if(control == FRAME1){
        //enviar RR0
        write_aux(rr0_message, 5);
        if(frame_num == 1){
                return 1;
        } else {
                return -4;
        }
        }
        return -5;
}


void debug_write(unsigned char *message, int numBytes){

        printf("\n%d bytes written\n", numBytes);

        printf("Sent: ");
        for(int i = 0; i < numBytes; i++){
        printf("0x%02X ", message[i]);
        }
        printf("\n");
```

```
}

void final_check(){
       if(state == STOP_BIG){
       if(control != FRAME1 && control != FRAME0){
              control = ERROR;
       }
       }
}


void display_statistics(){
       gettimeofday(&end, NULL);
       double time_taken = (end.tv_sec - start.tv_sec) + ((end.tv_usec -
start.tv_usec) / 1000000.0);
       double bit_rate = (bytes_received * 8) / time_taken;

       printf("\n--------------Statistics--------------\n");
       printf("Number of Frames with errors: %d\n", num_errors);
       printf("Time taken: %0.2f seconds\n", time_taken);
       printf("Number of bytes sent: %ld\n", bytes_received);
       printf("Total file size: %ld\n", bytes_sent);
       printf("Bit rate: %0.2f bps\n", bit_rate);
       printf("--------------------------------------\n");
}
```

application_layer.c

```
C/C++
// Application layer protocol implementation

#include "application_layer.h"
#include "application_layer_aux.h"
#include <stdio.h>
#include "link_layer.h"
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>

LinkLayer connectionParameters;
bool debug_application_layer = false;

void applicationLayer(const char *serialPort, const char *role, int
baudRate,
```

```
                    int nTries, int timeout, const char *filename){

        for (int i = 0; serialPort[i] != '\0'; i++) {
        strncat(connectionParameters.serialPort, &serialPort[i], 1);
        }

        connectionParameters.role = (strcmp(role, "tx") == 0) ? LlTx : LlRx;
        connectionParameters.baudRate = baudRate;
        connectionParameters.nRetransmissions = nTries;
        connectionParameters.timeout = timeout;

        //estabelecer a conecção
        if (llopen(connectionParameters) < 0) {
        perror("Error llopen");
        return;
        }
        else{
        printf("Connection established\n\n");
        }

        //transferir/receber o ficheiro
        if(connectionParameters.role == LlTx){
        // eviar o ficheiro
        if(sendFile(filename) == -1){
               perror("Error sending file applicationLayer\n");
               return;
        }
        }
        else{
        // receber o ficheiro
        if(receiveFile(filename) == -1){
               perror("Error receiving file applicationLayer\n");
               return;
        }
        }

        //fechar a conecção

        llclose(1);

        printf("\nConnection closed\n");

}

int sendFile(const char *filename){

        unsigned int control_frame_size = 0;
        FILE* my_file = fopen(filename, "rb");
```

```
        if(my_file == NULL){
        perror("Error opening file");
        return -1;
        }

        int pf = ftell(my_file);
        fseek(my_file, 0, SEEK_END);
        long int my_file_size = ftell(my_file) - pf;
        fseek(my_file, pf, SEEK_SET);

        unsigned char* control_frame_start = create_control_frame(1, filename,
my_file_size, &control_frame_size);

        printf("Writting the control word Start\n");

        if(llwrite(control_frame_start, control_frame_size) == -1){
        perror("Error sending control frame\n");
        return -1;
        }

        free(control_frame_start);

        //ler conteudo do ficheiro para um buffer
        unsigned char* file_content = (unsigned char*)malloc(sizeof(unsigned
char) * my_file_size);
        fread(file_content, sizeof(unsigned char), my_file_size, my_file);

        printf("Writting the file content\n\n");

        //enviar o ficheiro
        if(sendFileContent(file_content, my_file_size) == -1){
        perror("Error sending file content sendFile\n");
        return -1;
        }
        free(file_content);

        printf("Writting the control frame end\n\n");

        //enviar o control frame para terminar a conecção
        unsigned char* control_frame_end = create_control_frame(3, filename,
my_file_size, &control_frame_size);
        if(llwrite(control_frame_end, control_frame_size) == -1){
        perror("Error sending control frame\n");
        return -1;
        }
        free(control_frame_end);
```

```
	return 1;
}

int receiveFile(const char *filename){
	//receber o control frame para começar

	unsigned char* control_frame = (unsigned
char*)malloc(MAX_PAYLOAD_SIZE);
	bool flag = true;
	int res;

	printf("Reading the control word start\n\n");

	while(flag){
	res = llread(control_frame);
	if(res < 0){
		//lidar com os erros
		perror("Error receiving control frame\n");
		return -1;
	}
	else{
		//receber o control frame

		flag = false;
	}
	}

	free(control_frame);

	unsigned long int file_size = 0;
	int filename_size = 0;

	unsigned char* my_filename = receiveControlPacket(control_frame,
&file_size, &filename_size);

	if(debug_application_layer){
	debug_control_frame_rx(my_filename, filename_size, file_size);
	}

	FILE* my_file = fopen((char *) filename, "wb+");
	free(my_filename);

	unsigned char* data_frame_packet;
	unsigned char* data_frame;
	flag = true;
	int data_size = 0;
	int my_frame_num = 1;
```

```
	printf("Reading the file content\n\n");

	while(flag){
	data_frame_packet = (unsigned char*)malloc(MAX_PAYLOAD_SIZE);
	res = llread(data_frame_packet);
	if(res >= 0){
		//receber o data frame


		data_frame = receive_data_frame_packet(data_frame_packet, res,
&data_size);


		if(debug_application_layer){
		debug_print_frame(data_frame, data_size);
		}


		if(data_frame == NULL){
		printf("Reading the control word end\n\n");
		flag = false;
		}
		else{
		printf("Frame number: %d\n", my_frame_num);
		printf("Frame Accepted\n\n");
		my_frame_num++;
		fwrite(data_frame, sizeof(unsigned char), data_size, my_file);

		}
		free(data_frame_packet);
		free(data_frame);

	}
	}



	fclose(my_file);


	return 1;
}

unsigned char* create_control_frame(unsigned char c, const char* filename,
long int file_size, unsigned int* final_control_size){
	unsigned int pos = 0;

	unsigned char L1 = calculate_octets(file_size);
```

```
        unsigned char L2 = strlen(filename);
        *final_control_size = 5 + L1 + L2; //c + T1 + L1 + file_size + T2 + L2
+ file_name

        unsigned char *control_packet = (unsigned
char*)malloc(*final_control_size);

        control_packet[pos] = c;
        pos++;

        control_packet[pos] = 0;
        pos++;

        control_packet[pos] = L1;
        pos++;

        //colocar em big endian
        for (int i = 0; i < L1; i++) {
        control_packet[pos] = (file_size >> (8 * (L1 - i - 1))) & 0xFF;
        pos++;
        }

        control_packet[pos] = 1;
        pos++;

        control_packet[pos] = L2;
        pos++;

        for (int i = 0; i < L2; i++) {
        control_packet[pos] = filename[i];
        pos++;
        }

        if(debug_application_layer){
        debug_control_frame_tx(filename, L2, L1);
        }

        return control_packet;
}

unsigned char* receiveControlPacket(unsigned char* control_frame, unsigned
long int* file_size, int* filename_size){

        //File Size
        unsigned char L1 = control_frame[2];
        unsigned char file_size_bytes[L1];
```

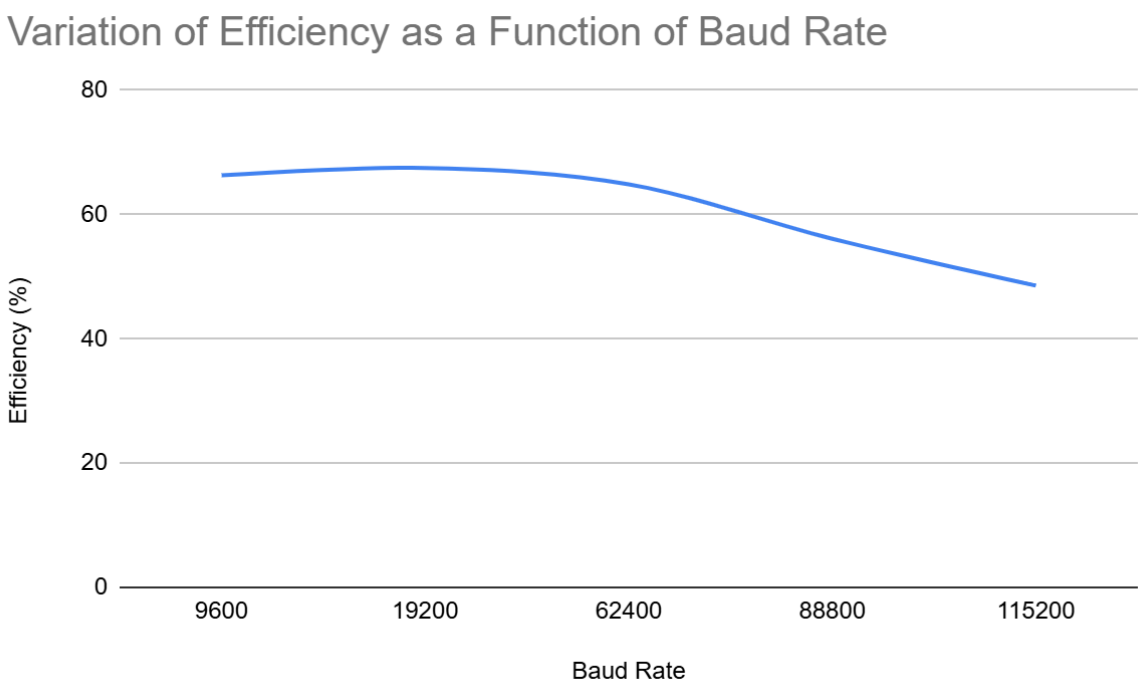
```
        for (int i = 0; i < L1; i++) {
        file_size_bytes[i] = control_frame[3 + i];
        }

        for(int i = 0; i < L1; i++){
        *file_size += file_size_bytes[i] << (8 * (L1 - i - 1));
        }

        //Filename
        unsigned char L2 = control_frame[4 + L1];
        unsigned char* filename = (unsigned char*)malloc(L2);

        for(int i = 0; i < L2; i++){
        filename[i] = control_frame[5 + L1 + i];
        }

        *filename_size = L2;

        return filename;
}

unsigned char calculate_octets(long int file_size) {
        unsigned char octets = 0;
        while (file_size > 0) {
        file_size >>= 8; // Shift right by 8 bits (1 byte)
        octets++;
        }
        return octets;
}


int sendFileContent(unsigned char* file_content, long int file_size){
        long int bytes_sent = 0;
        int sequence_number = 0;
        unsigned char* data_frame;
        unsigned char* data_frame_packet;
        int my_frame_num = 1;

        while(bytes_sent < file_size){
        int data_frame_size = (file_size - bytes_sent) > (MAX_PAYLOAD_SIZE -
4) ? (MAX_PAYLOAD_SIZE - 4) : (file_size - bytes_sent);

        data_frame = (unsigned char*)malloc(data_frame_size);
        memcpy(data_frame, file_content + bytes_sent, data_frame_size);

        int packet_size = 0;
        data_frame_packet = create_data_frame_packet(data_frame,
data_frame_size, &packet_size, sequence_number);
```

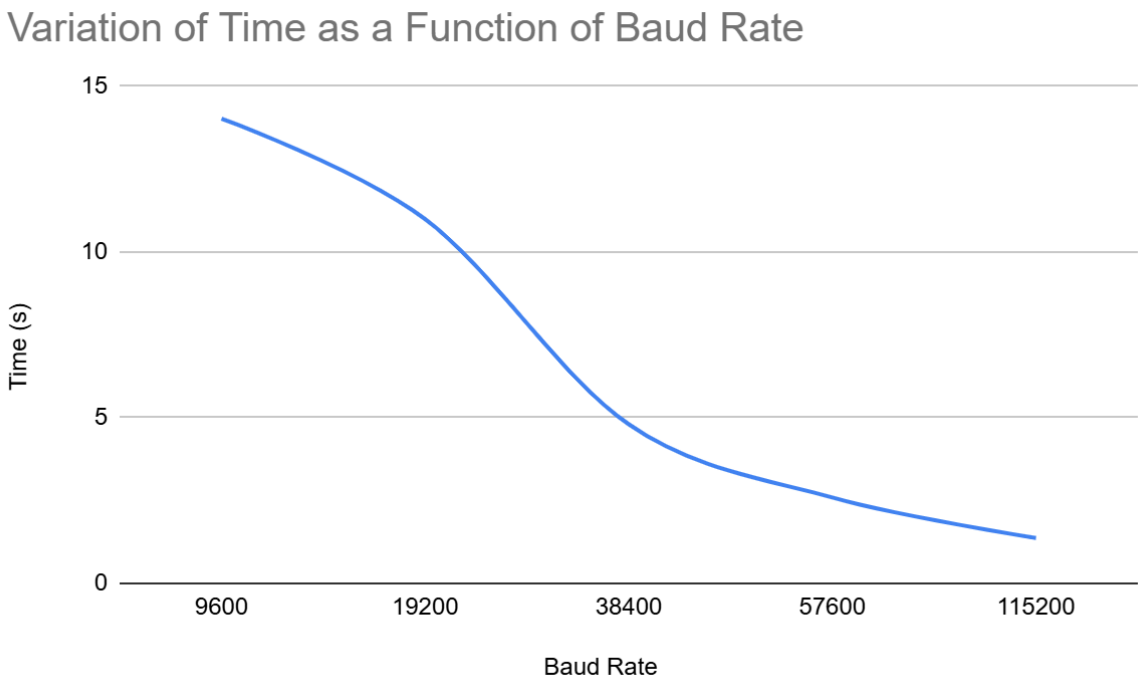
```
        printf("\n----------------------------------\n");
        printf("Frame number: %d\n", my_frame_num);
        printf("Frame size: %d\n", data_frame_size);

        if(llwrite(data_frame_packet, packet_size) == -1){
                perror("Error sending file content sendFileContent\n");
                return -1;
        }


        if(debug_application_layer){
                debug_print_frame(data_frame_packet, packet_size);
        }


        free(data_frame);
        free(data_frame_packet);

        bytes_sent += data_frame_size;
        sequence_number ++;
        my_frame_num++;
        }

        return 1;
}

unsigned char* create_data_frame_packet(unsigned char* data_frame, int
data_frame_size, int* packet_size, unsigned char sequence_number){
        *packet_size = data_frame_size + 4;
        unsigned char* packet = (unsigned char*)malloc(*packet_size);

        packet[0] = 2;
        packet[1] = sequence_number % 100;
        packet[2] = DataSizeL1(data_frame_size);
        packet[3] = DataSizeL2(data_frame_size);

        memcpy(packet + 4, data_frame, data_frame_size);

        return packet;
}

unsigned char* receive_data_frame_packet(unsigned char* data_frame_packet,
int data_frame_packet_size, int* data_size){
        if(data_frame_packet[0] != 2){
        return NULL;
        }
```

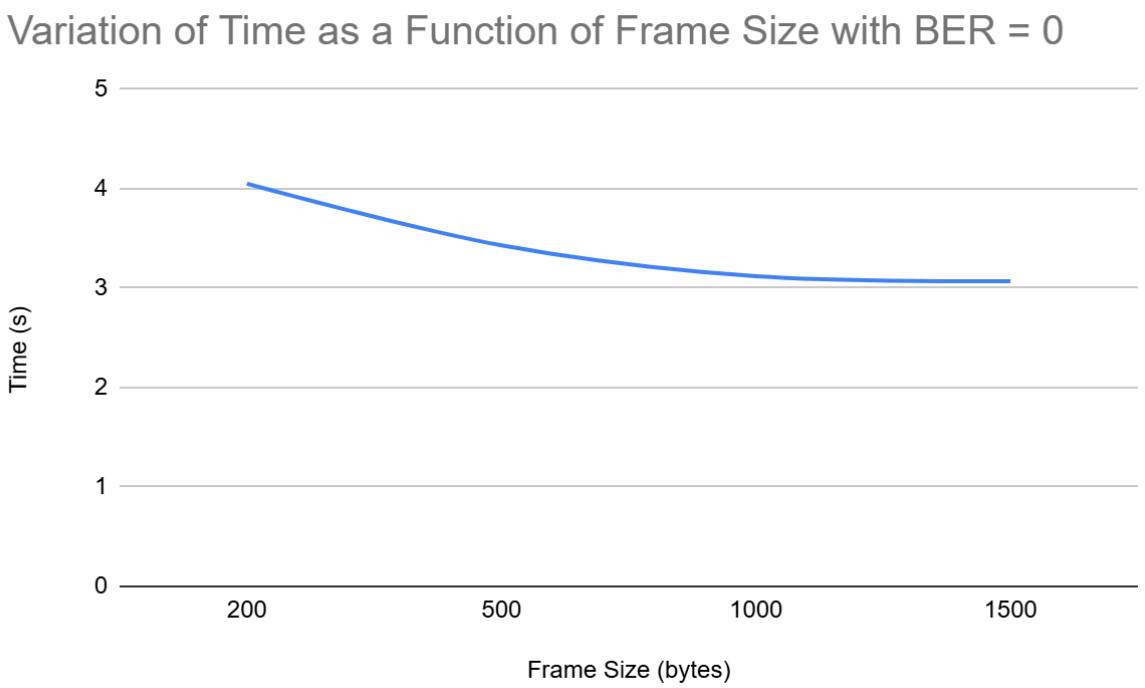
```
        unsigned char L1 = data_frame_packet[2];
        unsigned char L2 = data_frame_packet[3];

        *data_size = DATA_SIZE(L1, L2);

        unsigned char* data = (unsigned char*)malloc(*data_size);

        memcpy(data, data_frame_packet + 4, *data_size);

        return data;
}

void debug_control_frame_tx(const char* filename, unsigned int
filename_size, int size_of_file){
        if(filename == NULL){
        printf("Filename is NULL\n");
        return;
        }

        printf("\nFilename: ");
        for(int i = 0; i < filename_size; i++){
        printf("%c", filename[i]);
        }
        printf("\nFilename size: %d\n", filename_size);
        printf("Size of file: %d\n\n", size_of_file);

}


void debug_control_frame_rx(unsigned char* filename, unsigned int
filename_size, int size_of_file){
        if(filename == NULL){
        printf("Filename is NULL\n");
        return;
        }

        printf("\nFilename: ");
        for(int i = 0; i < filename_size; i++){
        printf("%c", filename[i]);
        }
        printf("\nFilename size: %d\n", filename_size);
        printf("Size of file: %d\n\n", size_of_file);
}

void debug_print_frame(unsigned char* frame, int frame_size){
        if(frame == NULL){
        printf("Frame is NULL\n");
        return;
```

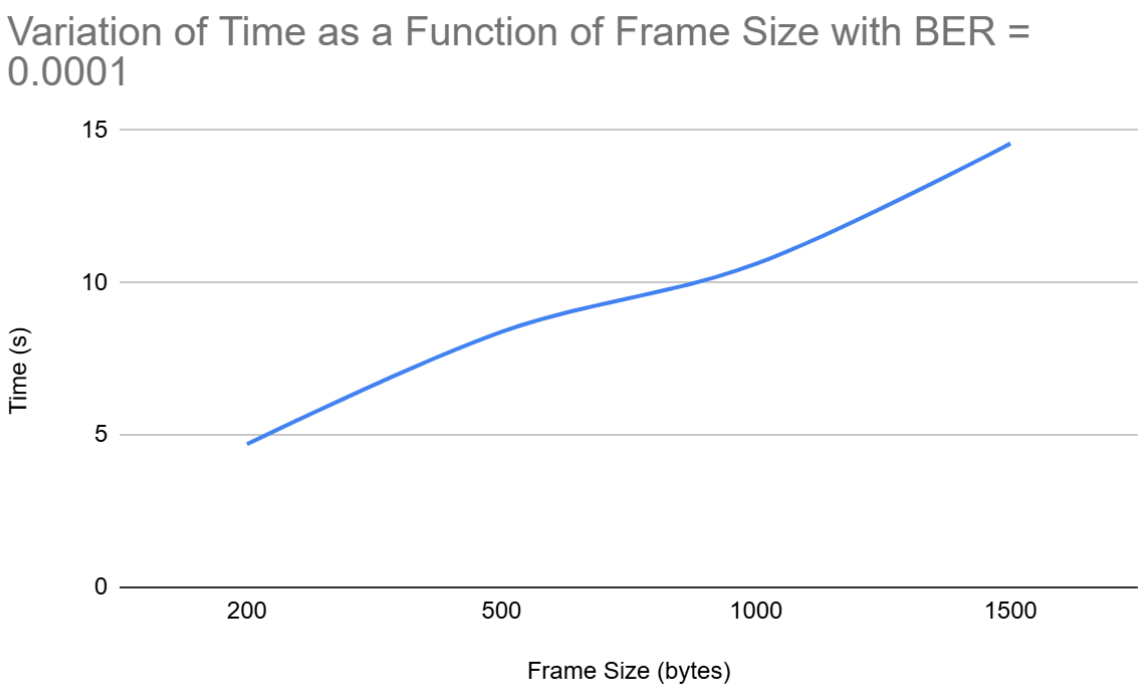
```
        }

        printf("\nFrame: ");
        for(int i = 0; i < frame_size; i++){
        printf("0x%02X ", frame[i]);
        }

        printf("\nFrame size: %d\n", frame_size);

        printf("\n\n");
}
```

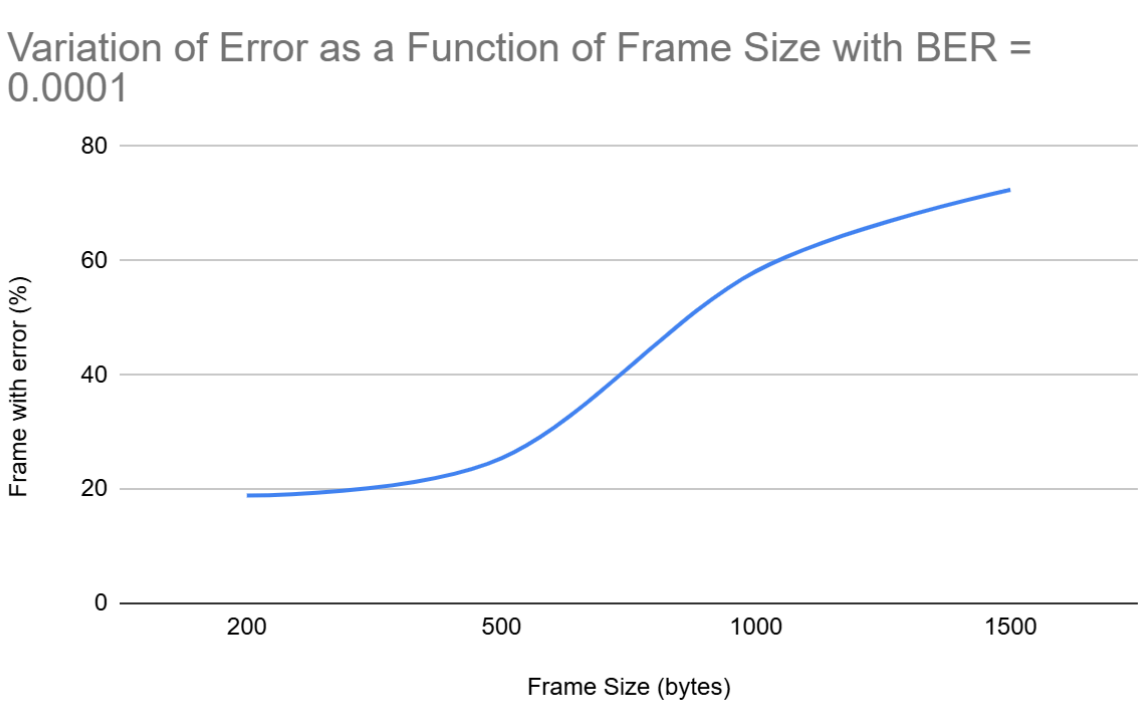
Appendix II - Graphs

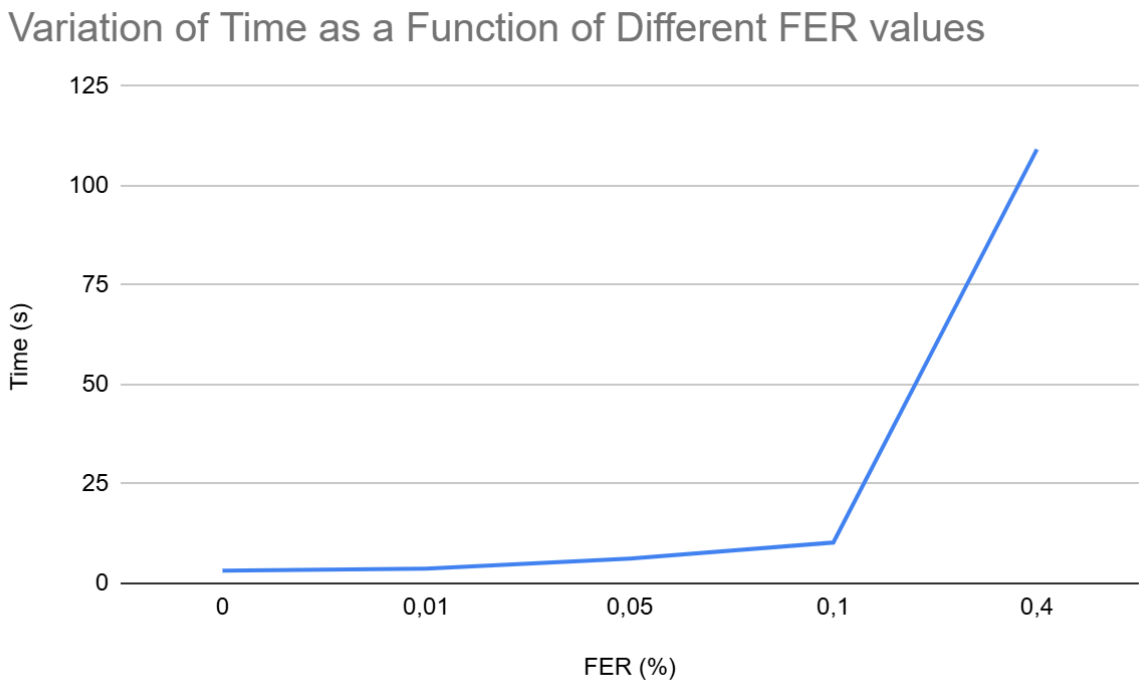



Variation of Time as a Function of Baud Rate
15
10
5
0
Time (s)
9600
19200
38400
57600
115200
Baud Rate


Variation of Time as a Function of Frame Size with BER = 0
5
4
3
2
1
0
Time (s)
200
500
1000
1500
Frame Size (bytes)


Variation of Time as a Function of Frame Size with BER = 0.0001
15
10
5
0
Time (s)
200
500
1000
1500
Frame Size (bytes)


Variation of Error as a Function of Frame Size with BER = 0.0001
80
60
40
20
0
Frame with error (%)
200
500
1000
1500
Frame Size (bytes)


Variation of Time as a Function of Different FER values
125
100
75
50
25
0
Time (s)
0
0,01
0,05
0,1
0,4
FER (%)