Data Link Protocol

RCOM - 1st Project

José Costa up2020048236@fe.up.pt
Mário Silva up202108841@fe.up.pt

November 11, 2024

Contents

1 Summary

This project aims to implement a robust data link protocol using a layered architecture, ensuring reliable file transmission over a serial port connection.

Through the development of this solution we were able to put theoretical concepts into practice, such as: framing; stuffing; error handling and Automatic Repeat reQuest (ARQ) Protocols, among others.

2 Introduction

The primary objective of this project was to develop a data transmission protocol utilizing the serial port. This report aims to clarify the functionality and implementation details of the developed protocol, highlighting both the theoretical foundations and practical applications.

The report is organized into the following sections:

- **Architecture** – Functional blocks and interfaces.
- **Code Structure** – APIs, main data structures, main functions, and their relationship with the architecture.
- **Main Use Cases** – Identification and sequences of function calls.
- **Logical Link Protocol** – Identification of the main functional aspects; description of the implementation strategy of these aspects with code excerpts.
- **Application Protocol** – Identification of the main functional aspects; description of the implementation strategy of these aspects with code excerpts.
- **Validation** – Description of the tests performed with a quantified presentation of the results, if possible.
- **Data Link Protocol Efficiency** – Statistical characterization of the protocol's efficiency, carried out using measurements on the developed code.
- **Conclusions** – Synthesis of the information presented in the previous sections and reflection on the achieved objectives and learnings.

3 Architecture

In terms of architecture, the application is divided into **two layers**: the **Application Layer** and the **Data Link Layer**. Additionally, there are **two operational modes**: **Transmitter** and **Receiver**. While the layers are represented as distinct code modules, the operational modes are not separate modules, instead, the same executable is used for both modes, with the user selecting the desired mode through command-line arguments. Communication with the serial port is facilitated by a provided dedicated **Serial Port API**, which abstracts low-level operations and allows the Data Link Layer to interact efficiently with the hardware.

- The **Application Layer** is responsible for reading from and writing to files, as well as creating and interpreting control and data packets. This layer coordinates transmission and reception operations by interacting directly with the Data Link Layer to send or receive information based on the selected operational mode.
- The **Data Link Layer** manages communication through the serial port, including frame construction, error detection and correction, and handling acknowledgments and rejections of packets. This layer relies on the **Serial Port API** to send and receive bytes, implementing link layer protocol mechanisms such as Automatic Repeat reQuest (ARQ) and frame stuffing to ensure data integrity and proper sequencing.

listings color

4 Code Structure

4.1 Application Layer

There was only need for one auxiliary data structures in this layer **struct packet**, the implemented functions are as follows:

```
// The packet structure encapsulates the size of the packet and a pointer to the actual data
typedef struct packet {
    int size;
    unsigned char* packet;
} packet;
// Main function of this block
void applicationLayer(const char *serialPort, const char *role, int baudRate, int nTries, int
    timeout, const char *filename);
// Function to get packet type as a string based on control field
const char* getPacketType(unsigned char control);
// Function to retrieve the next data packet from the file
struct packet nextpacket(FILE *fp, unsigned char p_num);
// Function to set up a control packet
struct packet setupControlPacket(int fileSize, enum PacketsControlField control);
// Function to read and interpret a control packet
int readControl(enum PacketsControlField control, struct packet p);
// Function to parse the next data packet and write it to the file
int parseNextPacket(struct packet p, FILE *fp, unsigned char seqNum);
```

4.2 Data Link Layer

In the this layer we use six auxiliary data structures, divided into three main groups: **Link Layer Role**, that identifies if the user is transmitter or receptor; **LinkLayer**, where we save the parameters associated with the current transfer and **States** structures, where we identify the current state of the frame interpreter state machine.

In the examples below, only one of the States structure is in display **OpenStates**, the other structure are similar to this one and can be viewed in the Annexes.

```
typedef enum
{
    LlTx,
    LlRx,
} LinkLayerRole;
typedef struct
{
    char serialPort[50];
    LinkLayerRole role;
    int baudRate;
    int nRetransmissions;
    int timeout;
} LinkLayer;
enum OpenStates
{
    OPEN_START,
    FLAG_OK,
    ADDR_OK,
    CTRL_OK,
    BCC_OK,
    OPEN_STOP
};
```

The implemented functions:

```
//Handles alarm signals to manage timeouts and retransmissions
void alarmHandler(int signal);
// Establishes a connection at the link layer by initializing serial port settings and
    performing the handshake process
int llopen(LinkLayer link_layer);
// Sends data frames over the established link layer connection, handling framing, error
    detection, and acknowledgment
int llwrite(const unsigned char *buf, int bufSize);
// Receives data frames from the link layer, ensuring data integrity and proper sequencing
int llread(unsigned char *packet, int packet_size);
// Terminates the link layer connection, performing necessary cleanup and final handshake
int llclose(int showStatistics);
// Determines whether to inject an error based on a specified error rate percentage
int should_inject_error(float error_rate_percentage);
// Corrupts a data frame by flipping a random bit to simulate transmission errors
void corrupt_frame(unsigned char *buf, int size);
```

5 Main Use Cases

5.1 Identification and Function Call Sequences

The program can be executed in 2 different modes: **Transmitter** and **Receiver**. This section outlines the primary use cases, detailing the function call sequences involved in establishing a connection, transmitting data, and terminating the connection for both roles.

5.1.1 Transmitter Flow

1. `main()`: Entry point of the application.
2. `applicationLayer()`: Determines role and initializes parameters.
3. `llopen()`: Opens the serial port and establishes the link layer connection.
4. File operations: Opens the file to be transmitted and obtains its size.
5. `setupControlPacket(START)`: Creates a START control packet containing the file size.
6. `llwrite(START packet)`: Sends the START control packet to the receiver.
7. **Loop over data packets**:
   (a) `nextpacket(file_pointer, p_num)`: Reads the next data packet from the file.
   (b) `llwrite(data packet)`: Sends the data packet to the receiver.
   (c) `p_num++`: Increments the packet sequence number.
8. `setupControlPacket(END)`: Creates an END control packet to signal the end of transmission.
9. `llwrite(END packet)`: Sends the END control packet to the receiver.
10. `llclose()`: Closes the link layer connection gracefully.

5.1.2 Receiver Flow

1. `main()`: Entry point of the application.
2. `applicationLayer()`: Determines and initializes parameters.
3. `llopen()`: Opens the serial port and waits for a connection from the transmitter.
4. `llread(datapacket, 1)`: Receives the START control packet.
5. `readControl(START, p)`: Parses the START packet to retrieve the file size.
6. File operations: Prepares the filename and opens the file for writing.
7. **Loop until all data received**:
   (a) `llread(datapacket, packet_size)`: Receives a data packet from the transmitter.
   (b) `parseNextPacket(p, file_pointer, p_num)`: Parses the data packet and writes it to the file.

   (c) `p_num++`: Increments the packet sequence number.

8. `llread(datapacket, 1)`: Receives the END control packet.
9. `readControl(END, p2)`: Parses the END packet and validates the file size.
10. `llclose()`: Closes the link layer connection gracefully.

6 Logical Link Protocol

This chapter focuses on the main functional aspects of the Logical Link Protocol (LLP) implementation and describes the strategies used in implementing these aspects.

6.1 Main Functional Aspects

The Logical Link Protocol is designed to provide reliable data transfer over a serial connection. The main functional aspects of the LLP implementation are:

1. **Connection Establishment** (`llopen`)
2. **Data Transmission** (`llwrite` and `llread`)
3. **Connection Termination** (`llclose`)
4. **Error Handling and Retransmission**

6.2 Implementation

6.2.1 Connection Establishment (`llopen`)

The `llopen` function is responsible for establishing a connection between the transmitter and receiver. It initializes the serial port and manages the handshake process using control frames.

Transmitter Role (`LlTx`):

- Sends a **SET** frame to initiate the connection.
- Waits for a **UA** (Unnumbered Acknowledgment) frame from the receiver.
- Implements a timeout and retransmission mechanism using alarms.
- Utilizes a state machine to parse the received frames.

Receiver Role (`LlRx`):

- Waits to receive a **SET** frame from the transmitter.
- Upon receiving the **SET** frame, sends back a **UA** frame to acknowledge.
- Uses a state machine to process the incoming **SET** frame.

6.2.2 Data Transmission (`llwrite` and `llread`)

Data transmission is handled by the `llwrite` function on the transmitter side and the `llread` function on the receiver side. These functions are responsible for sending and receiving data frames, respectively.

Data Framing:

- Frames consist of a start flag, address, control field, data, BCC (Block Check Character), and an end flag.
- Implements byte stuffing to escape special characters (`FLAG` and `ESC`).

Error Detection:

- Uses BCC for error detection on both control and data fields.
- The receiver checks the BCC to verify data integrity.

Acknowledgments:

- **RR** (Receiver Ready) frames are sent to acknowledge successful receipt.
- **REJ** (Reject) frames are sent if an error is detected.

Sequence Numbers:

- Sequence numbers are used to track frames and ensure correct ordering.

Error Injection for Testing:

- Simulates transmission errors to test error handling and retransmission mechanisms.
- Randomly decides whether to inject an error based on a predefined probability.
- Corrupts the frame by flipping a random bit.

6.2.3 Connection Termination (`llclose`)

The `llclose` function gracefully terminates the connection between the transmitter and receiver, ensuring that both parties are aware of the disconnection.

Transmitter Role (`LlTx`):

- Sends a **DISC** (Disconnect) frame to the receiver.
- Waits for a **DISC** frame in response.
- Sends a **UA** frame to acknowledge the disconnection.

Receiver Role (`llRx`):

- Waits for a **DISC** frame from the transmitter.
- Sends a **DISC** frame back to the transmitter.
- Waits for a **UA** frame to confirm the disconnection.

6.2.4 Error Handling and Retransmission

Error handling is a critical aspect of the LLP implementation, ensuring reliable data transfer even in the presence of transmission errors.

Implementation Strategy

- **Alarms and Timeouts:**
  - Alarms are used to detect timeouts when waiting for acknowledgments.
  - If a timeout occurs, the frame is retransmitted.
- **Frame Error Detection:**
  - BCC1 (Block Check Character) is used to detect errors in frames.
  - BCC2 is used to detect errors in the data.
  - If an error is detected, a **REJ** frame is sent, prompting retransmission.
- **Retransmissions:**
  - The number of retransmissions is limited by `nRetransmissions`.
  - The transmitter keeps track of retransmissions using `alarmCount`.
- **Artificial Error Injection:**
  - Errors are injected intentionally for testing purposes.
  - Functions are used to simulate errors by corrupting frames.

7 Application Protocol

This chapter focuses on the main functional aspects of the Application Protocol implementation and describes the strategies used in implementing these aspects.

7.1 Main Functional Aspects

The Application Protocol is designed to handle high-level communication tasks between the transmitter and receiver, particularly for file transfer operations. The main functional aspects of the Application Protocol implementation are:

1. **File Segmentation and Packaging**

2. **Control Packet Creation and Parsing**
3. **Data Packet Transmission and Reception**
4. **Error Handling and Data Integrity Verification**

7.2 Implementation

7.2.1 File Segmentation and Packaging

The application layer is responsible for reading the file to be transmitted and segmenting it into appropriately sized packets for transmission. This includes:

- **Reading the File:** Opening the file in binary mode and determining its size.
- **Creating Data Packets:** Dividing the file into chunks, each encapsulated in a data packet with a sequence number and size information.
- **Managing Sequence Numbers:** Using a sequence number (`p_num`) that increments with each packet to ensure correct ordering upon reception.

7.2.2 Control Packet Creation and Parsing

Control packets are used to signal the start and end of a file transfer. They contain metadata necessary for the receiver to properly handle the incoming data.

- **START Packet:** Created at the beginning of the transmission to inform the receiver about the file size and initiate the transfer process.
- **END Packet:** Sent after all data packets have been transmitted to signal the end of the file transfer.
- **Packet Structure:** Control packets include a control field indicating the packet type (`START` or `END`), a field specifying the parameter type (e.g., file size), and the actual parameter value.
- **Parsing Control Packets:** The receiver reads and interprets control packets to extract metadata, ensuring it is prepared to receive the data packets correctly.

7.2.3 Data Packet Transmission and Reception

Data packets carry the actual content of the file. The application layer handles their creation, transmission, reception, and storage.

- **Transmitter Side:**
  - Constructs data packets by reading chunks of the file and adding headers containing control information.
  - Uses the `llwrite` function to send data packets over the link layer.
  - Increments the sequence number after each packet is sent.
- **Receiver Side:**

- Uses the `llread` function to receive data packets from the link layer.
- Parses each data packet to extract the sequence number and data size.
- Writes the received data to the output file in the correct order.
- Increments the expected sequence number to match incoming packets.

7.2.4 Error Handling and Data Integrity Verification

To ensure reliable file transfer, the application layer includes mechanisms for error detection and handling.

- **Sequence Number Verification:** Checks if the sequence number of the received packet matches the expected value to detect out-of-order or missing packets.
- **Size Verification:** Validates that the size of the data received matches the size specified in the packet header.
- **Control Packet Validation:** Ensures that the control packets (`START` and `END`) are correctly formatted and contain valid information.
- **Error Reporting:** Prints error messages and handles exceptions if discrepancies are detected during transmission or reception.
- **Data Integrity:** Confirms that the total size of data received matches the expected file size, indicating a successful transfer.

8 Validation

Throughout the development the following tests were performed:

- Transfer of files of different sizes
- Transfer of files using different baud rates
- Partial and/or total interruption of the Serial Port
- Introduction of noise in the Serial Port

Furthermore, using the already mentioned functions **should_inject_error()** and **corrupt_frame()** artificial errors were introduced into the flow and dealt with appropriately by the solution.

The tests were successfully reproduced in the presence of the instructor during the class.

9 Efficiency

O nosso protocolo baseia-se num sistema *Stop-and-Wait ARQ*.

9.0.1 Definitions

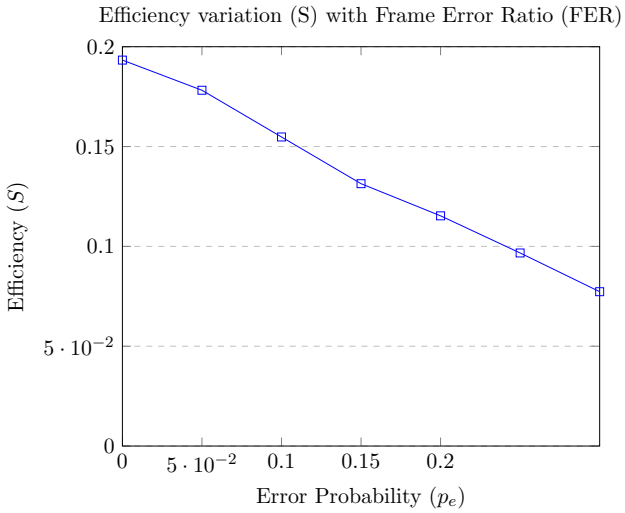
- **S** - efficiency
- **FER** ou $\boldsymbol{p_e}$ - error probability in a frame
- **R** ou $\boldsymbol{T_f}$ - received debit
- $\boldsymbol{T}_{\text{prop}}$ - frame propagation delay

9.0.2 Plots

The default values on our graphics are as follows:

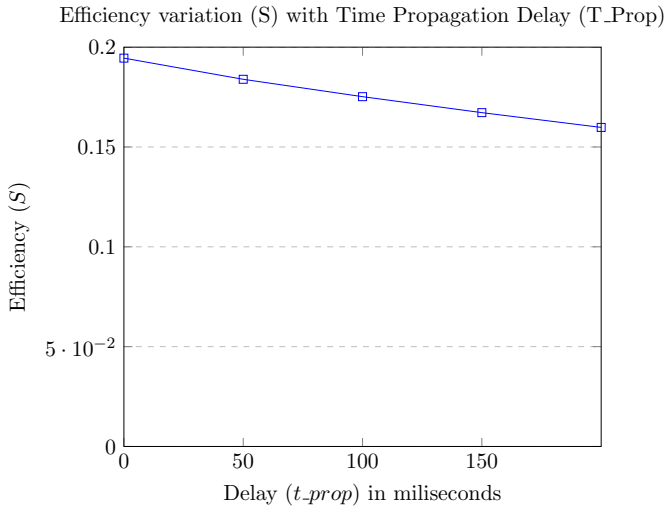
- $Baudrate = 9600$
- $Framesize = 250$ B
- $T_{\text{prop}} = 0$
- $FER = 0$

FER



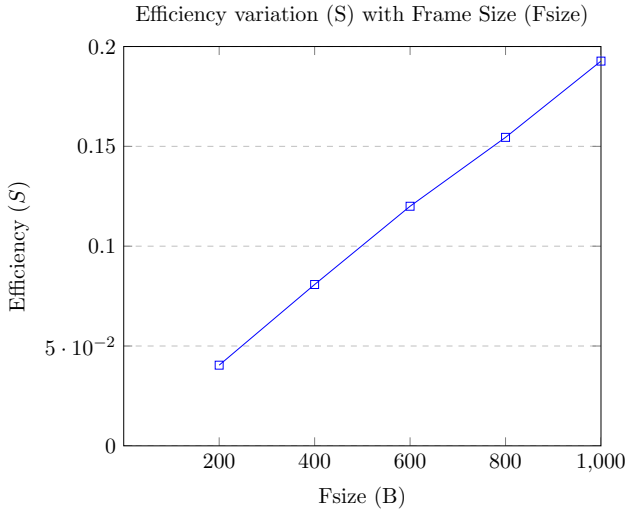
As observed, the efficiency of the protocol and the percentage of errors are inversely proportional. When a frame presents an error, it provokes a retransmission, which augments the transfer time, and, therefore, reduces the efficiency.

T_Prop



As observed, the efficiency of the protocol and the propagation delay are inversely proportional. With an added time for propagation , the efficiency naturally diminishes.

Frame Size



As observed, the efficiency of the protocol and the frame size are directly proportional, the smaller a frame is, the more transmissions are made, this augments the time of the protocol because each transmission has its associated overheads.

10 Conclusão

The Data Link Protocol was defined in 2 distinct layers that comunicated with each other, **LinkLayer** and **ApplicationLayer**. The **LinkLayer** also interacted with the provided Serial Port API, that simulated a real physical cable, therefore it needed to implement the necessary techniques for correctly dealing

with this types of data: stuffing, framing and error handling, among others. The **ApplicationLayer** was used as entry point for the solution and interacted directly with the file to be transfered.

The project served to better understand the above mentioned theoretical concepts in greater detail, as only practice can.

listings color

11 Annexes

Contents

- **application_layer**
- **link_layer**
- **constants**

The annexes below represent only the user defined code, files like: link_layer.h and application_layer.h, among others, were left unmodified, as per the teachers instructions.

11.1 application_layer.c

```
// Application layer protocol implementation

#include "application_layer.h"
#include "link_layer.h"

#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <termios.h>
#include <unistd.h>
#include "constants.h"
#include <sys/time.h>

// Color definitions using ANSI escape codes
#define RESET   "\033[0m"
#define RED     "\033[31m"
#define GREEN   "\033[32m"
#define YELLOW  "\033[33m"
#define BLUE    "\033[34m"
#define MAGENTA "\033[35m"
#define CYAN    "\033[36m"

// Function to get packet type as a string based on control field
const char* getPacketType(unsigned char control) {
    switch(control) {
        case START:
            return "START";
        case END:
            return "END";
```

```
        case DATA:
            return "DATA";
        default:
            return "UNKNOWN";
    }
}

typedef struct packet {
    int size;
    unsigned char* packet;
} packet;

struct packet nextpacket(FILE *fp, unsigned char p_num) {
    unsigned char* packetdata = (unsigned char*) malloc(PACKET_SIZE * sizeof(unsigned char));
    size_t read = 0;
    packet p;

    packetdata[0] = DATA;
    packetdata[1] = p_num % 255;

    if ((read = fread(packetdata + 4, 1, PACKET_SIZE - 4, fp)) < 0) {
        // printf("Error reading\n");
        printf(RED "[Error] Failed to read from file.%s\n", RESET);
    }
    packetdata[2] = (read & 0xFF00) >> 8;
    packetdata[3] = read & 0x00FF;

    p.size = read + 4;
    p.packet = packetdata;

    // printf("p.size: %d\n", p.size);
    printf(GREEN "[Packet] Created DATA packet, Size: %d bytes, Seq: %d%s\n", p.size, p_num,
    RESET);

    return p;
}

struct packet setupControlPacket(int fileSize, enum PacketsControlField control) {
    packet p;
    unsigned char* packetData = (unsigned char*) malloc(PACKET_SIZE * sizeof(unsigned char));

    packetData[0] = control;
    packetData[1] = 0x00;
    int num_bytes = 0;
    int copy_size = fileSize;

    while (copy_size != 0) {
        num_bytes += 1;
        copy_size = copy_size >> 8;
    }
```

```
    packetData[2] = num_bytes;

    for (int i = 1; i <= num_bytes; i++) {
        packetData[2 + i] = (fileSize >> (8 * (num_bytes - i)) & 0xff);
    }
    p.size = 3 + num_bytes;
    p.packet = packetData;

    // printf("Control Packet [%s] created, Size: %d bytes\n", getPacketType(control), p.size);
    printf(GREEN "[Control] [%s] packet created, Size: %d bytes%s\n", getPacketType(control),
    p.size, RESET);

    return p;
}

int readControl(enum PacketsControlField control, struct packet p) {
    unsigned char* packet = p.packet;

    if (control == START && packet[0] != START) return -1;
    if (control == END && packet[0] != END) return -1;
    if (control == DATA) return -1;

    if (packet[1] != 0) return -1;

    int k = packet[2];
    int filesize = 0;
    for (int i = 0; i < k; i++) {
        filesize = filesize << 8;
        filesize += packet[3 + i];
    }

    // printf("Parsed Control Packet [%s], File Size: %d bytes\n", getPacketType(control),
    filesize);
    printf(GREEN "[Control] [%s] packet parsed, File Size: %d bytes%s\n",
    getPacketType(control), filesize, RESET);

    return filesize;
}

int parseNextPacket(struct packet p, FILE *fp, unsigned char seqNum) {
    unsigned char* packet = p.packet;
    if (packet[0] != DATA) {
        // printf("Not data packet!\n");
        printf(RED "[Error] Received non-DATA packet.%s\n", RESET);
        return -1;
    }
    if (packet[1] != seqNum) {
        // printf("Not right packet sequence number!\n");
        printf(RED "[Error] Sequence number mismatch. Expected: %d, Received: %d%s\n", seqNum,
    packet[1], RESET);
        return -1;
```

```
    }

    // printf("%d, %d \n", packet[2], packet[3]);
    // Commented out as it's a debug statement
    // printf("Received Packet Size: %d bytes\n", (packet[2] * 256) + packet[3]);

    int packetsize = (packet[2] * 256) + packet[3];

    // Correct fwrite usage
    size_t bytesWritten = fwrite(packet + 4, 1, packetsize, fp);
    if (bytesWritten != packetsize) {
        // printf("\nCorrupted: %lu\n", fwrite(packet + 4, packetsize, 1, fp));
        printf(RED "[Error] Failed to write data to file. Expected: %d bytes, Written: %lu
    bytes%s\n", packetsize, bytesWritten, RESET);
        return -1;
    }

    // printf("\nPacket stored in file: %ld\nData in packet:", (long int) packetsize);
    printf(GREEN "[Packet] Stored %d bytes to file.%s\n", packetsize, RESET);
    /*
    for (int i = 0; i < packetsize; i++) printf("%x ", packet[4 + i]);
    printf("\n");
    */

    return packetsize;
}

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

    if (!strcmp("tx", role)) {
        linklayerrole = LlTx;
        // printf("Role: TRANSMITER\n");
        printf(GREEN "[Role] TRANSMITTER%s\n", RESET);
    } else if (!strcmp("rx", role)) {
        linklayerrole = LlRx;
        // printf("Role: RECEIVER\n");
        printf(GREEN "[Role] RECEIVER%s\n", RESET);
    } else {
        printf(RED "ERROR: Bad role\n%s", RESET);
        return;
    }

    LinkLayer options = {
        .role = linklayerrole,
        .baudRate = baudRate,
        .nRetransmissions = nTries,
        .timeout = timeout
    };
    strcpy(options.serialPort, serialPort);
```

```
int fd = llopen(options);
if (fd < 0) {
    printf(RED "[Error] llopen failed.%s\n", RESET);
    return;
}

unsigned char *datapacket;
unsigned char p_num = 0;
int res;

// printf("role: %d", linklayerrole);
printf(CYAN "[Info] Link Layer opened with role: %d%s\n", linklayerrole, RESET);

if (linklayerrole == LlTx) {
    int file_size = 0;
    FILE *file_pointer = NULL;

    file_pointer = fopen("penguin.gif", "rb");
    if (file_pointer == NULL) {
        printf(RED "File could not be opened.\n%s", RESET);
        return;
    }

    fseek(file_pointer, 0, SEEK_END);
    file_size = ftell(file_pointer);
    fseek(file_pointer, 0, SEEK_SET);

    // printf("File size: %d\n", file_size);
    printf(CYAN "[Info] File size: %d bytes%s\n", file_size, RESET);

    struct packet p = setupControlPacket(file_size, START);
    // printf("p.size: %d\n", p.size);
    // for (int i = 0; i < p.size; i++) printf("%x ", p.packet[i]);
    printf(CYAN "[Info] Sending START control packet.%s\n", RESET);
    llwrite(p.packet, p.size);
    free(p.packet); // Free allocated memory

    while (1) {
        p = nextpacket(file_pointer, p_num);
        datapacket = p.packet;
        int datapacketSize = p.size;

        if (datapacketSize > 4) {
            res = llwrite(datapacket, datapacketSize);
        } else {
            break;
        }
        free(datapacket);

        if (res == -1) {
```

```
                printf(RED "[Error] llwrite failed.%s\n", RESET);
                return;
            }

            p_num += 1;
        }

        fclose(file_pointer);

        p = setupControlPacket(file_size, END);
        printf(CYAN "[Info] Sending END control packet.%s\n", RESET);
        llwrite(p.packet, p.size);
        free(p.packet); // Free allocated memory

        if (llclose(0) != -1) {
            printf(GREEN "Done.\n%s", RESET);
        } else {
            printf(RED "Error closing connection.\n%s", RESET);
        }

    } else if (linklayerrole == LlRx) {
        int bits_received = 0;

        struct timeval ti, tf;
        double timeTaken;
        gettimeofday(&ti, NULL);

        datapacket = (unsigned char *) malloc(PACKET_SIZE * sizeof(unsigned char));

        res = llread(datapacket, 1);
        bits_received += res * 8;
        struct packet p = {res, datapacket};
        int tamanhofile = readControl(START, p);
        if (tamanhofile == -1) {
            printf(RED "Error reading START control packet.%s\n", RESET);
        }

        char *temp = (char*) malloc(50 * sizeof(char));
        sprintf(temp, filename, "-received%s");
        printf(GREEN "File name: %s%s\n", temp, RESET);

        FILE *file_pointer = NULL;
        file_pointer = fopen(temp, "wb");

        if (file_pointer == NULL) {
            perror("File not found.\n");
            free(temp);
            free(datapacket);
            return;
        }
```

```
    free(temp);

    int received = 0;
    while (received < tamanhofile) {
        if ((tamanhofile - received) / PACKET_SIZE == 0) {
            res = llread(datapacket, (tamanhofile - received) % PACKET_SIZE);
        } else {
            res = llread(datapacket, PACKET_SIZE);
        }
        p.size = res;
        p.packet = datapacket;
        res = parseNextPacket(p, file_pointer, p_num);
        received += res;
        bits_received += res * 8;
        p_num += 1;
    }

    if (received != tamanhofile) {
        printf(RED "Something went wrong in receiving\n%s", RESET);
    }

    fclose(file_pointer);

    res = llread(datapacket, 1);
    struct packet p2 = {res, datapacket};
    int file_size = readControl(END, p2);
    if (file_size == -1) {
        printf(RED "Error reading END control packet.%s\n", RESET);
    }
    if (file_size != tamanhofile) {
        printf(RED "Error in file sizes. Expected: %d, Received: %d%s\n", tamanhofile,
file_size, RESET);
    }

    gettimeofday(&tf, NULL);

    timeTaken = (tf.tv_sec - ti.tv_sec) * 1e6; // s to us
    timeTaken = (timeTaken + (tf.tv_usec - ti.tv_usec)) / 1e6; // us to s

    printf(CYAN "\n**** STATISTICS ****\n\n");
    printf(CYAN "Number of bits read = %d\n", bits_received);
    printf(CYAN "Time it took to send the file =  %fs\n", timeTaken);

    double R = bits_received / timeTaken;
    double S = R / baudRate;

    printf(CYAN "Baudrate = %lf\n", R);
    printf(CYAN "S = %lf\n\n", S);
    printf(CYAN "\n**********************\n\n");

    if (llclose(0) != -1) {
```

```
            printf(GREEN "Done.\n%s", RESET);
        } else {
            printf(RED "Error closing connection%s\n", RESET);
        }
    }
    free(datapacket);
}
```

11.2 link_layer.c

```
// Link layer protocol implementation

#include "link_layer.h"
#include "constants.h"

#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <signal.h>
#include <time.h>

// Color definitions using ANSI escape codes
#define RESET   "\033[0m"
#define RED     "\033[31m"
#define GREEN   "\033[32m"
#define YELLOW  "\033[33m"
#define BLUE    "\033[34m"
#define MAGENTA "\033[35m"
#define CYAN    "\033[36m"

// Function to decide whether to inject an error
int should_inject_error(float error_rate_percentage) {
    float rand_val = ((float)rand()) / RAND_MAX * 100.0; // Random float between 0 and 100
    return (rand_val < error_rate_percentage) ? 1 : 0;
}


// Function to corrupt a frame by flipping a random bit
void corrupt_frame(unsigned char *buf, int size) {
    if (size <= 0) return; // Nothing to corrupt

    int byte_index = rand() % size;            // Select a random byte
    int bit_index = rand() % 8;                // Select a random bit within the byte
    buf[byte_index] ^= (1 << bit_index);       // Flip the selected bit
```

```
}


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

int alarmEnabled = FALSE;
int alarmCount = 0;

// TODO: Refactor states
enum OpenStates
{
    OPEN_START,
    FLAG_OK,
    ADDR_OK,
    CTRL_OK,
    BCC_OK,
    OPEN_STOP
};

enum CloseStates
{
    CLOSE_START,
    CLOSE_FLAG_OK,
    CLOSE_ADDR_OK,
    DISC_OK,
    CLOSE_CTRL_OK,
    CLOSE_BCC_OK,
    CLOSE_STOP
};

enum Tx_State {
    T_Start,
    T_FLAG_RCV,
    T_ARECEIVED,
    T_CRECEIVED,
    T_BCC_OK,
    T_STOP_STATE
};

enum Rx_State {
    R_Start,
    R_FLAG_RCV,
    R_ARECEIVED,
    R_CRECEIVED,
    R_BCC_OK,
    R_STOP_STATE,
    R_ESCRECEIVED,
    R_ERROR
};
```

```c
// Alarm function handler
void alarmHandler(int signal)
{
    alarmEnabled = FALSE;
    alarmCount++;

    // printf("Alarm #%d\n", alarmCount);
}

// Frame error
volatile int FRAME_ERROR; // is dependent on something beyond our control

// Connection parameters
LinkLayer connectionParameters;

////////////////////////////////////////////
// LLOPEN
////////////////////////////////////////////
#include "serial_port.h"  // Include the serial_port header for API functions

int llopen(LinkLayer link_layer)
{
    // Copy link layer parameters to global connectionParameters
    strcpy(connectionParameters.serialPort, link_layer.serialPort);
    connectionParameters.role = link_layer.role;
    connectionParameters.baudRate = link_layer.baudRate;
    connectionParameters.nRetransmissions = link_layer.nRetransmissions;
    connectionParameters.timeout = link_layer.timeout;

    // Open the serial port using the serial_port API
    if (openSerialPort(connectionParameters.serialPort, connectionParameters.baudRate) < 0)
    {
        perror("openSerialPort failed");
        return -1;
    }

    unsigned char message[5] = { 0 }; // Adjusted to 5 bytes as per frame structure
    unsigned char response[5] = { 0 };
    unsigned char readChar;

    enum OpenStates state = OPEN_START;

    if (connectionParameters.role == LlTx) // TRANSMITTER
    {
        message[0] = FLAG;
        message[1] = A_Tx;
        message[2] = SET;
        message[3] = A_Tx ^ SET; // BCC1
        message[4] = FLAG;

        // Set the alarm function handler
```

```
(void)signal(SIGALRM, alarmHandler);

while (alarmCount < connectionParameters.nRetransmissions)
{
    if (alarmEnabled == FALSE)
    {
        // Send SET frame
        if (writeBytesSerialPort(message, 5) < 0) // Send only 5 bytes
        {
            perror("writeBytesSerialPort failed");
            closeSerialPort();
            return -1;
        }

        // Print sent SET frame
        printf(GREEN "[Sent] [%s] Tx -> Rx%s\n", "SET", RESET);

        alarmEnabled = TRUE;
        alarm(connectionParameters.timeout);

        // Wait cycle
        while (state != OPEN_STOP && alarmEnabled == TRUE)
        {
            usleep(PROPAGATION_DELAY);

            // Read a byte using readByteSerialPort instead of read
            int bytesRead = readByteSerialPort(&readChar);
            if (bytesRead == -1)
            {
                perror("readByteSerialPort failed");
                closeSerialPort();
                return -1;
            }
            else if (bytesRead == 0)
            {
                continue; // No byte read, continue waiting
            }

            // Process the byte based on the current state
            switch (state)
            {
            case OPEN_START:
                if (readChar == FLAG) state = FLAG_OK;
                break;
            case FLAG_OK:
                if (readChar == A_Tx)
                {
                    state = ADDR_OK;
                    response[1] = readChar;
                }
                else if (readChar != FLAG) state = OPEN_START;
```

```
                break;
            case ADDR_OK:
                if (readChar == FLAG) state = FLAG_OK;
                else if (readChar == UA)
                {
                    state = CTRL_OK;
                    response[2] = readChar;
                }
                else state = OPEN_START;
                break;
            case CTRL_OK:
                if (readChar == FLAG) state = FLAG_OK;
                else if (readChar == (response[1] ^ response[2])) // BCC = A ^ C
                {
                    response[3] = readChar;
                    state = BCC_OK;
                }
                else state = OPEN_START;
                break;
            case BCC_OK:
                if (readChar == FLAG)
                {
                    response[4] = FLAG;
                    state = OPEN_STOP;
                }
                else state = OPEN_START;
                break;
            default:
                break;
            }

            // Clear response if back to start
            if (state == OPEN_START)
            {
                memset(response, 0, 5);
            }
            else if (state == FLAG_OK)
            {
                memset(response, 0, 5);
                response[0] = FLAG;
            }
        }
    }

    // Stop if connection established
    if (state == OPEN_STOP)
    {
        printf(GREEN "[Received] [%s] Rx -> Tx%s\n", "UA", RESET);
        break;
    }
}
```

```
}
else if (connectionParameters.role == LlRx) // RECEIVER
{
    while (state != OPEN_STOP)
    {
        int bytesRead = readByteSerialPort(&readChar);
        if (bytesRead == -1)
        {
            perror("readByteSerialPort failed");
            closeSerialPort();
            return -1;
        }
        else if (bytesRead == 0)
        {
            continue; // No byte read, continue waiting
        }

        // Process the byte for RECEIVER role
        switch (state)
        {
        case OPEN_START:
            if (readChar == FLAG) state = FLAG_OK;
            break;
        case FLAG_OK:
            if (readChar == A_Tx)
            {
                state = ADDR_OK;
                message[1] = readChar;
            }
            else if (readChar != FLAG) state = OPEN_START;
            break;
        case ADDR_OK:
            if (readChar == FLAG) state = FLAG_OK;
            else if (readChar == SET)
            {
                state = CTRL_OK;
                message[2] = readChar;
            }
            else state = OPEN_START;
            break;
        case CTRL_OK:
            if (readChar == (A_Tx ^ SET))
            {
                message[3] = readChar;
                state = BCC_OK;
            }
            else if (readChar == FLAG) state = FLAG_OK;
            else state = OPEN_START;
            break;
        case BCC_OK:
            if (readChar == FLAG)
```

```
                {
                    message[4] = readChar;
                    state = OPEN_STOP;
                }
                else state = OPEN_START;
                break;
            default:
                break;
            }

            if (state == OPEN_START)
            {
                memset(message, 0, 5);
            }
            else if (state == FLAG_OK)
            {
                memset(message, 0, 5);
                message[0] = FLAG;
            }
        }

        // Print received SET frame
        printf(GREEN "[Received] [%s] Tx -> Rx%s\n", "SET", RESET);

        // Send UA response
        unsigned char ua_response[5] = { FLAG, A_Tx, UA, (A_Tx ^ UA), FLAG };

        if (writeBytesSerialPort(ua_response, 5) < 0) // Send only 5 bytes
        {
            perror("writeBytesSerialPort failed");
            closeSerialPort();
            return -1;
        }

        // Print sent UA frame
        printf(GREEN "[Sent] [%s] Rx -> Tx%s\n", "UA", RESET);
    }

    // Close port on connection failure due to maximum retries
    if (alarmCount >= connectionParameters.nRetransmissions)
    {
        printf(RED "[Error] Connection failed after %d retries.%s\n",
    connectionParameters.nRetransmissions, RESET);
        closeSerialPort();
        return -1;
    }

    printf(GREEN "CONNECTION ESTABLISHED!\n%s", RESET);
    return 0; // Connection established successfully
}
```

```
////////////////////////////////////////////
// LLWRITE
////////////////////////////////////////////

int llwrite(const unsigned char *packet, int bufSize)
{
    int frame_to_send = 1;
    unsigned char Data_bcc = 0;
    int fakebufferpos = 0;
    unsigned char buf[IBUF_SIZE] = {0};

    while (fakebufferpos < bufSize)
    {
        frame_to_send = !frame_to_send;
        unsigned char RR_ack = RR(!frame_to_send);
        unsigned char REJ_ack = REJ(frame_to_send);

        memset(buf, 0, IBUF_SIZE);
        Data_bcc = 0;

        // Set up frame header
        buf[0] = FLAG;
        buf[1] = A_Tx;
        buf[2] = CONTROL_IFRAME(frame_to_send);
        buf[3] = A_Tx ^ CONTROL_IFRAME(frame_to_send);

        int bufpos = 4;
        unsigned char tempbyte;

        // Frame stuffing
        for (int i = 0; i < DATA_BLOCK_SIZE; i++)
        {
            if (fakebufferpos + i >= bufSize) break;
            tempbyte = *(packet + fakebufferpos + i);
            Data_bcc ^= tempbyte;

            if (tempbyte == FLAG || tempbyte == ESC)
            {
                buf[bufpos++] = ESC;
                buf[bufpos++] = tempbyte ^ EXTRA_STUFFS;
            }
            else
            {
                buf[bufpos++] = tempbyte;
            }
        }

        fakebufferpos += DATA_BLOCK_SIZE;

        // Add Data BCC with stuffing if necessary
        if (Data_bcc == FLAG || Data_bcc == ESC)
```

```
{
    buf[bufpos++] = ESC;
    buf[bufpos++] = Data_bcc ^ EXTRA_STUFFS;
}
else
{
    buf[bufpos++] = Data_bcc;
}

buf[bufpos++] = FLAG; // Frame end

alarmCount = 0;
alarmEnabled = FALSE;
(void)signal(SIGALRM, alarmHandler);

// Transmission and acknowledgment wait loop
while (alarmCount < connectionParameters.nRetransmissions)
{
    if (alarmEnabled == FALSE)
    {

        int bytes_written;

        // **Error Injection Before Sending DATA Frame**
        if (should_inject_error(PROBABILITY_ERROR)) {
            // Preserve the original frame
            unsigned char buf_copy[IBUF_SIZE];
            memcpy(buf_copy, buf, bufpos);

            corrupt_frame(buf_copy, bufpos); // Corrupt the DATA frame
            printf(YELLOW "Artificial error injected into DATA frame.%s\n", RESET);


            // Send frame using writeBytesSerialPort
            bytes_written = writeBytesSerialPort(buf_copy, bufpos);
            if (bytes_written < 0)
            {
                perror("writeBytesSerialPort failed");
                return -1;
            }
        }else{
            // Send frame using writeBytesSerialPort
            writeBytesSerialPort(buf, bufpos);
            if (bytes_written < 0)
            {
                perror("writeBytesSerialPort failed");
                return -1;
            }
        }

        usleep(PROPAGATION_DELAY);
```

```
// Print sent I-frame
printf(GREEN "[Sent] [DATA] Tx -> Rx, Seq: %d%s\n", frame_to_send, RESET);

alarm(connectionParameters.timeout);
alarmEnabled = TRUE;
sleep(1); // Wait for transmission

// Wait for acknowledgment
unsigned char readChar;
FRAME_ERROR = FALSE;
enum Tx_State state = T_Start;
unsigned char commandbuff[TRANSMITTER_READ_BUFF_SIZE] = {0};

while (alarmEnabled == TRUE && state != T_STOP_STATE)
{
    int bytesRead = readByteSerialPort(&readChar);
    if (bytesRead == -1)
    {
        perror("readByteSerialPort failed");
        return -1;
    }
    else if (bytesRead == 0)
    {
        continue; // No byte read, continue waiting
    }

    // Process acknowledgment byte by byte
    switch (state)
    {
    case T_Start:
        if (readChar == FLAG)
            state = T_FLAG_RCV;
        break;
    case T_FLAG_RCV:
        if (readChar == A_Tx)
        {
            state = T_ARECEIVED;
            commandbuff[1] = readChar;
        }
        else if (readChar != FLAG)
        {
            state = T_Start;
        }
        break;
    case T_ARECEIVED:
        if (readChar == FLAG)
        {
            state = T_FLAG_RCV;
        }
        else if (readChar == REJ_ack || readChar == RR_ack)
```

```
        {
            state = T_CRECEIVED;
            commandbuff[2] = readChar;
        }
        else
        {
            state = T_Start;
        }
        break;
    case T_CRECEIVED:
        if (readChar == (commandbuff[1] ^ commandbuff[2]))
        {
            commandbuff[3] = readChar;
            state = T_BCC_OK;
        }
        else if (readChar == FLAG)
        {
            state = T_FLAG_RCV;
        }
        else
        {
            state = T_Start;
        }
        break;
    case T_BCC_OK:
        if (readChar == FLAG)
        {
            commandbuff[4] = FLAG;
            state = T_STOP_STATE;
        }
        else
        {
            state = T_Start;
        }
        break;
    default:
        break;
    }

    // Reset if back to start
    if (state == T_Start)
    {
        memset(commandbuff, 0, TRANSMITTER_READ_BUFF_SIZE);
    }
    else if (state == T_FLAG_RCV)
    {
        memset(commandbuff, 0, TRANSMITTER_READ_BUFF_SIZE);
        commandbuff[0] = FLAG;
    }
    else if (state == T_STOP_STATE && commandbuff[2] == REJ_ack)
    {
```

```
                    FRAME_ERROR = TRUE;
                }
            }

            // Handle acknowledgment
            if (state == T_STOP_STATE)
            {
                if (commandbuff[2] == RR_ack)
                {
                    printf(GREEN "[Received] [RR] Rx -> Tx, Seq: %d%s\n", frame_to_send,
RESET);
                    break; // Frame acknowledged successfully
                }
                else if (commandbuff[2] == REJ_ack)
                {
                    printf(RED "[Received] [REJ] Rx -> Tx, Seq: %d%s\n", frame_to_send,
RESET);
                    FRAME_ERROR = TRUE;
                }
            }

            // Retransmit if frame error occurred
            if (FRAME_ERROR == TRUE)
            {
                printf(RED "[Error] Frame %d corrupted. Retransmitting...%s\n",
frame_to_send, RESET);
                alarmCount = 0;
                continue;
            }
        }

        // Exceeded retransmission limit, close port
        if (alarmCount >= connectionParameters.nRetransmissions)
        {
            printf(RED "[Error] Exceeded maximum retransmissions for frame %d%s\n",
frame_to_send, RESET);
            closeSerialPort();
            return -1;
        }
    }
}
    return 1; // Full data successfully sent
}

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

int llread(unsigned char *packet, int packet_size)
{
    int frame_number = 0;
```

```
int packetposition = 0;
unsigned char messagebuffer[5] = {0};
unsigned char frame[DATA_BLOCK_SIZE] = {0};
unsigned char controlbuffer[6] = {0};

while (packetposition < packet_size)
{
    memset(frame, 0, DATA_BLOCK_SIZE);
    memset(controlbuffer, 0, 6);
    int framepos = 0;
    unsigned char data_bcc = 0x0;
    unsigned char frame_required = CONTROL_IFRAME(frame_number);
    unsigned char prev_frame = CONTROL_IFRAME(!frame_number);

    enum Rx_State state = R_Start;
    unsigned char readChar;

    while (state != R_STOP_STATE && state != R_ERROR)
    {
        int bytesRead = readByteSerialPort(&readChar);
        if (bytesRead == -1)
        {
            perror("readByteSerialPort failed");
            return -1;
        }
        else if (bytesRead == 0)
        {
            continue; // No byte read, continue waiting
        }

        // Process the byte according to the current state
        switch (state)
        {
        case R_Start:
            if (readChar == FLAG)
                state = R_FLAG_RCV;
            break;
        case R_FLAG_RCV:
            if (readChar == A_Tx)
            {
                state = R_ARECEIVED;
                controlbuffer[1] = readChar;
            }
            else if (readChar != FLAG)
            {
                state = R_Start;
            }
            break;
        case R_ARECEIVED:
            if (readChar == FLAG)
            {
```

```
            state = R_FLAG_RCV;
        }
        else if (readChar == frame_required || readChar == prev_frame)
        {
            state = R_CRECEIVED;
            controlbuffer[2] = readChar;
        }
        else
        {
            state = R_Start;
        }
        break;
    case R_CRECEIVED:
        if (readChar == (controlbuffer[1] ^ controlbuffer[2]))
        {
            state = R_BCC_OK;
        }
        else if (readChar == FLAG)
        {
            state = R_FLAG_RCV;
        }
        else
        {
            state = R_Start;
        }
        break;
    case R_BCC_OK:
        if (readChar == FLAG)
        {
            state = R_STOP_STATE;
        }
        else if (readChar == ESC)
        {
            state = R_ESCRECEIVED;
        }
        else
        {
            frame[framepos++] = readChar;
            data_bcc ^= readChar;
        }
        break;
    case R_ESCRECEIVED:
        state = R_BCC_OK;
        if ((readChar ^ EXTRA_STUFFS) == FLAG || (readChar ^ EXTRA_STUFFS) == ESC)
        {
            frame[framepos++] = readChar ^ EXTRA_STUFFS;
            data_bcc ^= readChar ^ EXTRA_STUFFS;
        }
        else
        {
            state = R_ERROR;
```

```
            }
            break;
        default:
            break;
        }

        // Reset control and frame if starting over
        if (state == R_Start)
        {
            memset(frame, 0, DATA_BLOCK_SIZE);
            memset(controlbuffer, 0, 6);
            framepos = 0;
            data_bcc = 0x0;
        }
        else if (state == R_FLAG_RCV)
        {
            memset(controlbuffer, 0, 6);
            memset(frame, 0, DATA_BLOCK_SIZE);
            controlbuffer[0] = FLAG;
            data_bcc = 0x0;
            framepos = 0;
        }
        else if (state == R_STOP_STATE)
        {
            framepos--; // Exclude the Data BCC byte from the data frame
            if (data_bcc != 0x00)
            {
                state = R_ERROR;
            }
        }
    }

    // Prepare response based on frame status
    messagebuffer[0] = FLAG;
    messagebuffer[1] = A_Tx;
    messagebuffer[4] = FLAG;

    if (state == R_ERROR)
    {
        messagebuffer[2] = (controlbuffer[2] == frame_required) ? REJ(frame_number) :
RR(frame_number);
        printf(RED "[Received] [Corrupted I] Rx -> Tx, Seq: %d%s\n", frame_number, RESET);
        printf(RED "[Sent] [%s] Tx -> Rx, Seq: %d%s\n", "REJ", frame_number, RESET);
    }
    else
    {
        if (controlbuffer[2] == frame_required)
        {
            memcpy(packet + packetposition, frame, framepos);
            packetposition += framepos;
            frame_number = !frame_number;
```

```
                printf(GREEN "[Received] [DATA] Rx -> Tx, Seq: %d%s\n", frame_number, RESET);
            }
            messagebuffer[2] = RR(frame_number);
            printf(GREEN "[Sent] [RR] Tx -> Rx, Seq: %d%s\n", frame_number, RESET);
        }

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

        // Send acknowledgment (RR or REJ) using writeBytesSerialPort
        if (writeBytesSerialPort(messagebuffer, 5) < 0)
        {
            perror("writeBytesSerialPort failed");
            return -1;
        }
        sleep(1); // Wait for acknowledgment to send
    }

    return packetposition; // Number of bytes received and stored in packet
}


////////////////////////////////////////////
// LLCLOSE
////////////////////////////////////////////

int llclose(int showStatistics)
{
    unsigned char message[5] = { FLAG, A_Tx, DISC, (A_Tx ^ DISC), FLAG };
    unsigned char readChar;

    if (connectionParameters.role == LlTx) // TRANSMITTER
    {
        // Set up the alarm function handler
        alarmCount = 0;
        alarmEnabled = FALSE;
        (void)signal(SIGALRM, alarmHandler);

        // Transmission and acknowledgment wait loop for DISC
        while (alarmCount < connectionParameters.nRetransmissions)
        {
            if (alarmEnabled == FALSE)
            {
                // Send DISC using writeBytesSerialPort
                if (writeBytesSerialPort(message, 5) < 0) // Send only 5 bytes
                {
                    perror("writeBytesSerialPort failed");
                    closeSerialPort();
                    return -1;
                }

                // Print sent DISC frame
```

```
printf(GREEN "[Sent] [%s] Tx -> Rx%s\n", "DISC", RESET);

alarmEnabled = TRUE;
alarm(connectionParameters.timeout);
sleep(1); // Wait for transmission

// Wait for DISC acknowledgment
enum CloseStates state = CLOSE_START;
unsigned char rx_response[5] = {0};

while (alarmEnabled == TRUE && state != CLOSE_STOP)
{
    int bytesRead = readByteSerialPort(&readChar);
    if (bytesRead == -1)
    {
        perror("readByteSerialPort failed");
        closeSerialPort();
        return -1;
    }
    else if (bytesRead == 0)
    {
        continue; // No byte read, continue waiting
    }

    // Process received DISC acknowledgment byte by byte
    switch (state)
    {
    case CLOSE_START:
        if (readChar == FLAG)
            state = CLOSE_FLAG_OK;
        break;
    case CLOSE_FLAG_OK:
        if (readChar == A_Tx)
        {
            state = CLOSE_ADDR_OK;
            rx_response[1] = readChar;
        }
        else if (readChar != FLAG)
        {
            state = CLOSE_START;
        }
        break;
    case CLOSE_ADDR_OK:
        if (readChar == FLAG)
        {
            state = CLOSE_FLAG_OK;
        }
        else if (readChar == DISC)
        {
            state = DISC_OK;
            rx_response[2] = readChar;
```

```
            }
            else
            {
                state = CLOSE_START;
            }
            break;
        case DISC_OK:
            if (readChar == (rx_response[1] ^ rx_response[2]))
            {
                rx_response[3] = readChar;
                state = CLOSE_BCC_OK;
            }
            else if (readChar == FLAG)
            {
                state = CLOSE_FLAG_OK;
            }
            else
            {
                state = CLOSE_START;
            }
            break;
        case CLOSE_BCC_OK:
            if (readChar == FLAG)
            {
                rx_response[4] = readChar;
                state = CLOSE_STOP;
            }
            else
            {
                state = CLOSE_START;
            }
            break;
        default:
            break;
        }
    }

    if (state == CLOSE_STOP)
    {
        // Print received DISC acknowledgment
        printf(GREEN "[Received] [%s] Rx -> Tx%s\n", "DISC", RESET);

        // Prepare and send UA response after receiving DISC
        unsigned char ua_message[5] = { FLAG, A_Tx, UA, (A_Tx ^ UA), FLAG };
        if (writeBytesSerialPort(ua_message, 5) < 0) // Send only 5 bytes
        {
            perror("writeBytesSerialPort failed");
            closeSerialPort();
            return -1;
        }
```

```
                // Print sent UA frame
                printf(GREEN "[Sent] [%s] Tx -> Rx%s\n", "UA", RESET);
                break;
            }
        }
    }
}
else if (connectionParameters.role == LlRx) // RECEIVER
{
    enum CloseStates state = CLOSE_START;

    // Wait for DISC from transmitter
    while (state != CLOSE_STOP)
    {
        int bytesRead = readByteSerialPort(&readChar);
        if (bytesRead == -1)
        {
            perror("readByteSerialPort failed");
            closeSerialPort();
            return -1;
        }
        else if (bytesRead == 0)
        {
            continue; // No byte read, continue waiting
        }

        // Process DISC frame
        switch (state)
        {
        case CLOSE_START:
            if (readChar == FLAG)
                state = CLOSE_FLAG_OK;
            break;
        case CLOSE_FLAG_OK:
            if (readChar == A_Tx)
            {
                state = CLOSE_ADDR_OK;
                message[1] = readChar;
            }
            else if (readChar != FLAG)
            {
                state = CLOSE_START;
            }
            break;
        case CLOSE_ADDR_OK:
            if (readChar == FLAG)
            {
                state = CLOSE_FLAG_OK;
            }
            else if (readChar == DISC)
            {
```

```
                state = DISC_OK;
                message[2] = readChar;
            }
            else
            {
                state = CLOSE_START;
            }
            break;
        case DISC_OK:
            if (readChar == (message[1] ^ message[2]))
            {
                state = CLOSE_BCC_OK;
                message[3] = readChar;
            }
            else if (readChar == FLAG)
            {
                state = CLOSE_FLAG_OK;
            }
            else
            {
                state = CLOSE_START;
            }
            break;
        case CLOSE_BCC_OK:
            if (readChar == FLAG)
            {
                message[4] = readChar;
                state = CLOSE_STOP;
            }
            else
            {
                state = CLOSE_START;
            }
            break;
        default:
            break;
        }
    }

    // Print received DISC frame
    printf(GREEN "[Received] [%s] Tx -> Rx%s\n", "DISC", RESET);

    // Send DISC in response
    if (writeBytesSerialPort(message, 5) < 0) // Send only 5 bytes
    {
        perror("writeBytesSerialPort failed");
        closeSerialPort();
        return -1;
    }

    // Print sent DISC frame
```

```
printf(GREEN "[Sent] [%s] Rx -> Tx%s\n", "DISC", RESET);
sleep(1); // Wait for transmission

// Wait for UA from transmitter
alarmCount = 0;
alarmEnabled = FALSE;
(void)signal(SIGALRM, alarmHandler);
while (alarmCount < connectionParameters.nRetransmissions)
{
    if (alarmEnabled == FALSE)
    {
        alarm(connectionParameters.timeout);
        alarmEnabled = TRUE;

        enum CloseStates state = CLOSE_START;
        unsigned char response_ua[5] = {0};

        while (alarmEnabled == TRUE && state != CLOSE_STOP)
        {
            int bytesRead = readByteSerialPort(&readChar);
            if (bytesRead == -1)
            {
                perror("readByteSerialPort failed");
                closeSerialPort();
                return -1;
            }
            else if (bytesRead == 0)
            {
                continue; // No byte read, continue waiting
            }

            // Process UA response byte by byte
            switch (state)
            {
            case CLOSE_START:
                if (readChar == FLAG)
                    state = CLOSE_FLAG_OK;
                break;
            case CLOSE_FLAG_OK:
                if (readChar == A_Tx)
                {
                    state = CLOSE_ADDR_OK;
                    response_ua[1] = readChar;
                }
                else if (readChar != FLAG)
                {
                    state = CLOSE_START;
                }
                break;
            case CLOSE_ADDR_OK:
                if (readChar == FLAG)
```

```
                {
                    state = CLOSE_FLAG_OK;
                }
                else if (readChar == UA)
                {
                    state = CLOSE_CTRL_OK;
                    response_ua[2] = readChar;
                }
                else
                {
                    state = CLOSE_START;
                }
                break;
            case CLOSE_CTRL_OK:
                if (readChar == (response_ua[1] ^ response_ua[2]))
                {
                    state = CLOSE_BCC_OK;
                    response_ua[3] = readChar;
                }
                else if (readChar == FLAG)
                {
                    state = CLOSE_FLAG_OK;
                }
                else
                {
                    state = CLOSE_START;
                }
                break;
            case CLOSE_BCC_OK:
                if (readChar == FLAG)
                {
                    response_ua[4] = FLAG;
                    state = CLOSE_STOP;
                }
                else
                {
                    state = CLOSE_START;
                }
                break;
            default:
                break;
            }
        }

        if (state == CLOSE_STOP)
        {
            // Print received UA frame
            printf(GREEN "[Received] [%s] Rx -> Tx%s\n", "UA", RESET);
            break; // UA received successfully
        }
    }
```

```
        }

        // Restore terminal settings and close the port
        if (closeSerialPort() < 0)
        {
            perror("closeSerialPort failed");
            return -1;
        }

        return 1; // Disconnection successful
    }

    return 0;
}
```

11.3 constants.h

```
// constants.h

#ifndef CONSTANTS_H
#define CONSTANTS_H

#define SET 0x03
#define DISC 0x0B
#define UA 0x07//Unnumbered ACK
#define RR(n) 0x05 | (n<<7)//Positive ACK
#define REJ(n) 0x01 |(n<<7)//Negative ACK

#define FLAG 0x7E
#define A_Tx 0x03 //sent by Tx or reply from Rx
#define A_Rx 0x01 //sent by Rx or reply from Tx
//BCC1 A ^ C

#define CONTROL_IFRAME(n) (n<<6)  //Info frame control byte (n=0or1)
//BCC2 Data1 ^ Data2 ^ Data3 ^...

#define ESC 0x7D
#define EXTRA_STUFFS 0x20


enum PacketsControlField
{
   START = 0x02,
   DATA = 0x01,
   END = 0x03

};
#define BUF_SIZE 5 //{F,A,C,BCC1,F}
```

```
#define DATA_BLOCK_SIZE 200
#define IBUF_SIZE ((2 * DATA_BLOCK_SIZE) + 6) //data block + control info
#define TRANSMITTER_READ_BUFF_SIZE 5
#define PACKET_SIZE (4 * DATA_BLOCK_SIZE)  //max payload

//STATISTICS TESTING VALUES
#define KMS 10000
#define T_PROP_PER_KM 5 //us - 5 microseconds per km ? check this

#define PROPAGATION_DELAY 0

#define PROBABILITY_ERROR 0 //percentage of error probability


#endif // CONSTANTS_H
```