Faculdade de Engenharia da Universidade do Porto

Data Link Protocol

Instructor:

Eduardo Nuno Moreira Soares de Almeida

Redes de Computadores

Turma 15 – Grupo 2

- Clarisse Maria Teixeira de Carvalho (up202008444@fc.up.pt)

- Maria Eduarda Pacheco Mendes Araújo (up202004473@fc.up.pt)

Porto, 31 de outubro de 2024

Summary:

This project was carried out as part of the Computer Networks course and focuses on implementing a data communication protocol for file transfer over na RS-232 serial port.

foi realizado no âmbito da Unidade Curricular de Redes de Computadores

Introduction:

The goal of this project is to implement a data link layer protocol based on the provided specifications. This protocol enables a transmitter and receiver to transfer files stored on the hard disk between two computers connected via na RS-232 serial cable.

We developed and tested a data link protocol, in line with the specifications provided, for file transfers through a serial port.

The report is divided as follows:

- **Architecture:** Functional blocks and interfaces.
- **Code Structure:** APIs, main data structures, key functional and their relation to the architecture.
- **Main Use Cases:** Identification of core project functionalities, including function cal sequences.
- **Logical Link Protocol:** Logical connection functionality and implementation strategies.
- **Application Protocol:** Application layer functionality and implementation strategies.
- **Validation:** Tests performed to assess implementation correctness.
- **Data Link Protocol Efficiency:** Evaluation of the Stop & Wait protocol efficiency in the data link layer.
- **Conclusions:** Summary of information presented in previous sections and refletion on learning outcomes.
- **Appendix I - Source Code**

Architecture:

- Functional Blocks:

The project is structured around two primary layers: the **Link Layer** and the **Application Layer**

- *LinkLayer*: This layer is responsible for implementing the data link protocol and handling low-level communication over the serial port. Found in link_layer.h and link_layer.c, it manages essential tasks such as establishing and terminating connections, framing data, error detection, and handling acknowledgments. Key functions in this layer include llopen() for connection setup, llwrite() for data transmission, and llclose() for disconnection.

- Application Layer: Implemented in application_layer.h and application_layer.c, this layer provides higher-level functionality to facilitate file transfer operations. It leverages the Link Layer API to send and receive data packets through the serial port, handling file preparation and ensuring reliable transmission. Users interact with this layer to set parameters such as frame size, transfer speed, and retry limits.

- Interfaces:

The program is designed to operate on two computers, each connected via an RS-232 serial cable. It uses two terminals—one on each computer—to enable communication between a **transmitter** and a **receiver**. Each terminal runs the program in a specified mode.

```
[sissi@archlinux code]$ ./bin/main /dev/ttyS11 9600 rx penguin-received.gif
Starting link-layer protocol application
  - Serial port: /dev/ttyS11
  - Role: rx
  - Baudrate: 9600
  - Number of tries: 3
  - Timeout: 4
  - Filename: penguin-received.gif
------------------------------------
Start program. Please wait...
------------------------------------
Connection Successfully
------------------------------------
Bytes read: 24
Bytes read: 994
Bytes read: 994
Bytes read: 994
Bytes read: 994
Bytes read: 994
Bytes read: 994
Bytes read: 994
Bytes read: 994
Bytes read: 994
Bytes read: 994
Bytes read: 994
Bytes read: 82
Bytes read: 24
Disconnecting receiver...
Message sent
Closing connection...
------------------------------------
```

```
[sissi@archlinux code]$ ./bin/main /dev/ttyS10 9600 tx penguin.gif
Starting link-layer protocol application
  - Serial port: /dev/ttyS10
  - Role: tx
  - Baudrate: 9600
  - Number of tries: 3
  - Timeout: 4
  - Filename: penguin.gif
------------------------------------
Role stored as: Transmitter (LlTx).
Start program. Please wait...
Message sent
------------------------------------
Connection Successfully
------------------------------------
Message sent
Positive Acknowledgement :)
------------------------------------
Bytes uccessfully written: 30
Message sent
Positive Acknowledgement :)
------------------------------------
Bytes uccessfully written: 1006
Sent packet: 0
Message sent
Positive Acknowledgement :)
------------------------------------
Bytes uccessfully written: 1007
Sent packet: 1
Message sent
Positive Acknowledgement :)
```

- **Transmitter Mode**: Sends data frames to the receiver, ensuring they are correctly received and acknowledged.
- **Receiver Mode**: Receives, validates, and stores data frames, acknowledging successful reception or requesting retransmission if errors are detected.

Estrutura do código:

LinkLayer:

Two auxiliary data structures were used in this layer: LinkLayer, where the parameters associated with data transfer are characterized, and LinkLayerRole, which identifies whether the computer is a transmitter or receiver

---

```
typedef enum
{
    LlTx,
    LlRx,
} LinkLayerRole;

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

```
// Open a connection using the "port" parameters defined in struct linkLayer.
// Return "1" on success or "-1" on error.
int llopen(LinkLayer connectionParameters);

// Send data in buf with size bufSize.
// Return number of chars written, or "-1" on error.
int llwrite(const unsigned char *buf, int bufSize);

// Receive data in packet.
// Return number of chars read, or "-1" on error.
int llread(unsigned char *packet);

// Close previously opened connection.
// if showStatistics == TRUE, link layer should print statistics in the console on close.
// Return "1" on success or "-1" on error.
int llclose(int showStatistics);
```

There were the functions implemented

ApplicationLayer

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

In implementing this layer, there was no need to create auxiliary data structures

---

Alarm

```
void reset_alarm_count();
void alarm_handler(int signal);
int get_alarm_count();
int get_alarm_flag();
void set_alarm_flag(int flag);
```

These were the functions implemented

Serial Port

```
int openSerialPort(const char *serialPort, int baudRate);
int closeSerialPort();
int readByteSerialPort(unsigned char *byte);
int writeBytesSerialPort(const unsigned char *bytes, int numBytes);
```

There were the functions implemented

State

```
typedef enum
{
    COMMAND_SET,
    COMMAND_DISC,
    COMMAND_DATA,
    RESPONSE_UA,
    RESPONSE_REJ
} command;

typedef enum
{
    TRANSMITTER,
    RECEIVER
} role;

typedef enum {
    RESPONSE_NULL,
    UA,
    PA_F0,
    PA_F1,
    REJ_F0,
    REJ_F1
} response;
```

```
typedef enum {
    START_STATE,        //
    FLAG_RCV,           //
    ADDRESS_RCV,        //
    CONTROL_RCV,        //
    DATA_RCV,           //
    BCC_VER,            //
    FINAL_STATE         //
} state;

typedef struct  {
    state curr_state;
    uint8_t address;
    uint8_t control;
    response prev_response;
    role curr_role;
    command curr_command;
} state_machine;
```

```
//GETTERS
state get_curr_state();
role get_curr_role();
command get_curr_command();
response get_prev_response();

//SETTERS
void set_state(state s);
void set_role(role r);
void set_command(command c);
```

```
/**
 * @brief processes a received byte based on the current s
 * @param byte
 */
void update_state(unsigned char byte);

/**
 * @brief resets the state machine to its initial state
 */
void reset_state();
```

Main use cases:

The program can be run in transmitter and receiver modes. The functions to be used and the sequence of calls will differ depending on the choice made.

- Transmiter:

1. **Connection Setup (llopen)**:
    - The transmitter initiates the connection by calling llopen(), which configures the serial port parameters.
    - Then, start_transmissor() sends a SET supervisory frame to request a connection.
    - The transmitter waits for an acknowledgment (UA) from the receiver. If it receives the UA frame, the connection is established; otherwise, it retries.
2. **Data Transmission (llwrite)**:
    - The transmitter reads the data to be sent and prepares an I-frame, including a sequence number and error-checking BCC fields.
    - The frame is then "stuffed" to escape special characters before transmission.
    - Using transmit_information_frame(), the frame is sent, and the transmitter waits for an acknowledgment.
    - If the acknowledgment is positive (matching sequence number), the transmitter proceeds with the next frame; if negative, it retries up to a maximum of three attempts.
3. **Disconnection (llclose)**:
    - Once the file transfer is complete, the transmitter initiates disconnection by sending a DISC frame.
    - After receiving a DISC acknowledgment, it sends a UA frame to confirm termination and then closes the serial port

- Receiver:

The program can be run in transmitter and receiver modes. The functions to be used and the sequence of calls will differ depending on the choice made.

1. **Connection Setup (llopen)**:
    - The receiver waits for a SET frame from the transmitter.
    - Upon receiving it, the receiver responds with a UA frame, establishing the connection.
2. **Data Reception (llread)**:
    - The receiver waits for incoming I-frames.
    - For each received frame, the llread() function destuffs the data and checks the BCC fields for errors.
    - If the data is valid and the sequence number matches, the receiver stores the data and sends a positive acknowledgment (PA). If an error is detected, the receiver sends a rejection (REJ) frame.
3. **Disconnection (llclose)**:
    - Upon receiving a DISC frame from the transmitter, the receiver responds with a DISC frame.
    - Finally, it waits for a UA acknowledgment before closing the serial port connection.

Logical Link Protocol:

The data link layer in this project serves as the interface for direct communication between the transmitter and receiver via the serial port, implementing a Stop-and-Wait protocol to ensure reliable data transfer. This layer manages the setup, data transmission, error-checking, and termination of the communication.

- **Connection Establishment**:
  The connection is initialized by the **llopen()** function, which configures the serial port and initiates communication. In transmitter mode, **llopen()** sends a SET (setup) supervisory frame to request a connection. The receiver responds with a UA (unnumbered acknowledgment) supervisory frame, signaling successful connection establishment. Upon receiving the UA frame, the transmitter proceeds with data transmission.

- **Data Transmission**:
  File data is transmitted frame-by-frame using the **llwrite()** function. Each data frame includes sequence numbers and error-checking fields (BCC1 and BCC2) for data integrity. The transmitter waits for an acknowledgment (ACK) from the receiver after each frame is sent. If the receiver detects an error (e.g., due to transmission noise or corruption), it sends a REJ (reject) frame instead, prompting the transmitter to retransmit the affected frame. The Stop-and-Wait mechanism ensures only one frame is sent at a time, providing reliable, ordered delivery.

- **Data Reception**:
  On the receiver side, data is processed by the **llread()** function, which reads each incoming frame from the serial port. It first removes any "stuffed" escape sequences (added to avoid misinterpreting special characters) and then validates BCC1 and BCC2. If validation succeeds and the sequence number matches, the receiver sends a positive acknowledgment, allowing the transmitter to proceed with the next frame. If validation fails, a REJ frame is sent to request retransmission.

- **Connection Termination**:
  Once all frames are successfully transferred, the transmitter terminates the session by invoking **llclose()**, which sends a DISC (disconnect) frame to initiate the termination process. The receiver responds with a DISC frame, and upon acknowledgment with a final UA frame, the serial port is closed, completing the file transfer.

This structure ensures that each stage of communication—from setup to transmission, error handling, and disconnection—follows a reliable, organized protocol that can effectively handle data errors and connection disruptions.

Application Protocol:

The application layer performs three main tasks: managing file operations, creating and interpreting control and data packets, and coordinating the transmission with the data link layer. The functions are described below.

Main Functions:

1. **processReceivedPacket()**: Interprets received packets and manages file operations:
   - Handles three packet types: DATA, START CONTROL, and END CONTROL.
   - For DATA packets, verifies size and writes data to the destination file.
   - For START and END CONTROL packets, opens and closes the destination file, respectively.

2. **createStartControlPacket()** and **createEndControlPacket()**: Generate control packets with metadata for the file being transferred:
    - Encapsulate file information, such as size and name, into a start control packet.
    - The end control packet signifies the completion of the file transfer.
3. **createDataPayloadPacket()**: Forms data packets for file content:
    - Adds packet metadata, such as sequence number and data length, ensuring error-checking mechanisms.
4. **applicationLayer()**: Manages the transfer process and coordinates with the link layer:
    - **Transmitter Mode**: Prepares file data, creates packets, and transmits them using llwrite().
    - **Receiver Mode**: Receives packets, validates them, and reconstructs the file using llread() and processReceivedPacket().

Transmission Flow:

1. **Connection Setup**: llopen() initializes the serial connection.
2. **File Transfer**: The transmitter reads the file, packetizes it, and sends it sequentially. The receiver validates and stores received packets.
3. **Disconnection**: llclose() closes the connection after transfer completion.

Validation:

During the project, several tests were conducted to evaluate the robustness and accuracy of the implemented data link protocol, especially under adverse conditions. These tests were designed to simulate real-world scenarios where data communication may experience interruptions, noise, or errors.

1. **Partial and/or Total Interruption of the Serial Port**:
    - **Objective**: To test how the protocol responds to sudden loss of connection, which may occur due to hardware disconnection or failure.
    - **Procedure**: During file transfer, the serial port connection was intentionally interrupted at random intervals. We monitored how the protocol handled these interruptions, including any reconnection attempts and error messages logged.
    - **Expected Outcome**: The protocol should detect the loss of connection, attempt reconnection (up to the maximum retry limit), and log an appropriate error if reconnection fails.
2. **Introduction of Noise via a Short Circuit**:
    - **Objective**: To test the protocol's resilience against electrical interference, which is common in real-world environments.
    - **Procedure**: A short circuit was introduced in the serial port during data transmission to simulate noise. We observed the receiver's response, checking if it could detect and handle corrupted frames.
    - **Expected Outcome**: The protocol should detect data errors through checksum verification (BCC fields) and request retransmission of corrupted frames, ensuring data integrity.

3. **Rejection of Data Frames by Introducing Random Errors**:
    - **Objective**: To test the protocol's handling of random transmission errors and its ability to manage frame rejection and retransmission.
    - **Procedure**: Errors were randomly injected into data frames to trigger rejections by the receiver. We observed if the receiver sent a REJ (reject) frame and if the transmitter properly retransmitted the affected frames.
    - **Expected Outcome**: The protocol should correctly reject corrupted frames, log the errors, and handle retransmission attempts until the data is accurately received or the maximum retries are reached

Pending Validation

Although these tests were conducted independently, they have not yet been reviewed in the presence of the instructor. A formal presentation in the laboratory is planned, where further feedback and validation may be obtained.

Data Link Protocol Efficiency:

File size = 85600 bits

Baud rate 9600 – time = 14 s

85600/14 = 6 114,2866114,2857/9600 = 0,637

Baud rate 4800 – time = 25 s

85600 / 25 = 3 424

3424 / 4800 = 0,713 = 71,3%

Baud rate 56000 – time = 3 s

85600 / 3 = 28 533,333

28533 / 56000 = 0,51 = 51%

- With higher baud rates, efficiency tends to be lower (51%) due to factors such as proportionally higher overhead and the need for rapid processing.

- On the other hand, for lower baud rates, efficiency tends to increase (71.3%), as the overhead and intervals between packets represent a smaller proportion of the total transmission time.

Conclusions:

With this project we were able to consolidate what we had learned in theory classes. This project made us understand that *LinkLayer* was responsible for interacting with the serial port and managing the information webs and that *ApplpicationLayer* was responsible for the layer that interacted directly with the file that was going to be transferred.

Appendix I - Source Code.

Alarm.c

```
#include "alarm.h"

// alarm_flag represents if the alarm is enabled or not

int alarm_flag = FALSE;

int alarm_count = 0;

void reset_alarm_count()

{

   alarm_count = 0;

}

void set_alarm_flag(int flag)

{

   alarm_flag = flag;

}

int get_alarm_count()

{

   return alarm_count;

}

int get_alarm_flag()

{

   return alarm_flag;

}

// Alarm function handler

void alarm_handler(int signal)

{

   set_alarm_flag(TRUE);

   alarm_count++;
```

```
    printf("Alarm #%d\n", alarm_count);
}
```

Application_layer.c

```
// Application layer protocol implementation
#include "application_layer.h"

int processReceivedPacket(unsigned char *buffer, int buffer_size, const char *file_path)
{
    static int dest_file_fd;
    switch (buffer[0])
    {
    // DATA PACKET
    case 1:
        if (buffer_size < 4) {
            printf("Invalid data packet size\n");
            return -1;
        }

        unsigned data_size = buffer[3] + (256 * buffer[2]);

        if (buffer_size < data_size + 4) {
            printf("Data packet size mismatch\n");
            return -1;
        }

        if (write(dest_file_fd, &buffer[4], data_size) < 0) {
            perror("Error writing to destination file");
            return -1;
        }
        return 0;

    // START CONTROL PACKET
    case 2:
        if ((dest_file_fd = open(file_path, O_WRONLY | O_CREAT | O_TRUNC, 0777)) < 0) {
            perror("Error opening destination file");
            return -1;
        }
        return 0;

    // END CONTROL PACKET
    case 3:
        if (close(dest_file_fd) < 0) {
            perror("Error closing destination file");
            return -1;
        }
        // RESET FILE DESCRIPTOR
        dest_file_fd = -1;
        return 0;
    default:
        printf("Invalid packet received\n");
```

```
        return -1;
    }
}

unsigned char *createStartControlPacket(const char *filename, struct stat *file_stat, int l1, int l2) {
    int start_size = 5 + l1 + l2;
    unsigned char *packet = (unsigned char *)malloc(start_size);

    if (packet == NULL) {
        return NULL;
    }

    // INIT WITH 0
    memset(packet, 0, start_size);

    // PACKET DETAILS
    packet[0] = 2;          // PACKET ID
    packet[1] = 0;          // CONTROL INFO
    packet[2] = l1;         // LEN FILE
    memcpy(&packet[3], &(file_stat->st_size), l1);
    packet[3 + l1] = 1;     // FILE NAME
    packet[4 + l1] = l2;    // FILE LEN
    memcpy(&packet[5 + l1], filename, l2);

    return packet;
}

unsigned char *createEndControlPacket(const char *filename, struct stat *file_stat, int l1, int l2) {
    int start_size = 5 + l1 + l2;
    unsigned char *endpacket = (unsigned char *)malloc(start_size);

    if (endpacket == NULL) {
        return NULL;
    }

    // PACKET DETAILS
    endpacket[0] = 3;          // ID PACKET
    endpacket[1] = 0;          // CONTROL INFO
    endpacket[2] = l1;         // LEN FILE
    memcpy(&endpacket[3], &(file_stat->st_size), l1);
    endpacket[3 + l1] = 1;     // FILE NAME
    endpacket[4 + l1] = l2;    // FILE LEN
    memcpy(&endpacket[5 + l1], filename, l2);

    return endpacket;
}

unsigned char *createDataPayloadPacket(const unsigned char *msg, ssize_t bytes_read, unsigned packet_number) {
    int packet_size = bytes_read + 4;
    unsigned char *packet = (unsigned char *)malloc(packet_size);

    if (packet == NULL) {
        return NULL;
    }
```

```
    packet[0] = 1;                      // DATA ID
    packet[1] = packet_number % 256;    // PACKET NUMBER
    packet[2] = (bytes_read >> 8) & 0xFF;   // LEN FILE HIGH BYTE
    packet[3] = bytes_read & 0xFF;      // LEN FILE LOW BYTE
    memcpy(&packet[4], msg, bytes_read);

    return packet;
}

void applicationLayer(const char *serialPort, const char *role, int baudRate,
                      int nTries, int timeout, const char *filename)
{
    LinkLayer connectionParameters;
    strncpy(connectionParameters.serialPort, serialPort, sizeof(connectionParameters.serialPort) - 1);
    connectionParameters.serialPort[sizeof(connectionParameters.serialPort) - 1] = '\0';

    printf("-----------------------------------\n");

    if (strcmp(role, "tx") == 0) {
        connectionParameters.role = LlTx;  // Transmitter
        printf("Role stored as: Transmitter (LlTx).\n");

    } else if (strcmp(role, "rx") == 0) {
        connectionParameters.role = LlRx;  // Receiver
    } else {
        perror("Error: Role not stored correctly.\n");
        exit(-1);
    }

    connectionParameters.baudRate = baudRate;
    connectionParameters.nRetransmissions = nTries;
    connectionParameters.timeout = timeout;

    if (llopen(connectionParameters) < 0){
        perror("llopen failed.\n");
        exit(-1);
    }

    if (connectionParameters.role == LlTx) {
        int file_fd;
        struct stat file_stat;

        if (stat(filename, &file_stat) < 0) {
            perror("Error getting file information.\n");
            exit(-1);
        }

        if ((file_fd = open(filename, O_RDONLY)) < 0) {
            perror("Error opening file.\n");
            exit(-1);
        }

        int l1 = sizeof(file_stat.st_size);
```

```
int l2 = strlen(filename);
unsigned char *packet = createStartControlPacket(filename, &file_stat, l1, l2);
if (!packet) {
    fprintf(stderr, "Failed to create data packet.\n");
    close(file_fd);
    exit(-1);
}

if (llwrite(packet, 5 + l1 + l2) < 0) {
    perror("llwrite failed.\n");
    free(packet);
    close(file_fd);
    exit(-1);
}

free(packet);

unsigned char msg[MAX_PAYLOAD_SIZE - 6];
unsigned packet_number = 0;

while (1) {
    ssize_t bytes_read = read(file_fd, msg, MAX_PAYLOAD_SIZE - 10);
    if (bytes_read < 0) {
        perror("Error reading file\n");
        exit(-1);
    }

    // END FILE
    if (bytes_read == 0) {
        break;
    }

    unsigned char *data_packet = createDataPayloadPacket(msg, bytes_read, packet_number);
    if (!data_packet) {
        fprintf(stderr, "Failed to create data packet\n");
        close(file_fd);
        exit(-1);
    }

    if (llwrite(data_packet, bytes_read + 4) < 0) {
        perror("llwrite failed for data packet\n");
        free(data_packet);
        exit(-1);
    }

    printf("Sent packet: %d\n", packet_number);
    packet_number++;
    free(data_packet);
}

// SEND END PACKET
unsigned char *end_packet = createEndControlPacket(filename, &file_stat, l1, l2);
if (!end_packet) {
    fprintf(stderr, "Failed to create end packet\n");
```

```
        close(file_fd);
        exit(-1);
    }

    end_packet[0] = 3;
    if (llwrite(end_packet, 5 + l1 + l2) < 0) {
        fprintf(stderr, "llwrite failed for end packet\n");
    }

    free(end_packet);
    close(file_fd);
}
// Receiver: Receive the file
else if (connectionParameters.role == LlRx) {
    unsigned char buf[MAX_PAYLOAD_SIZE - 6] = {0};

    int bytesRead = llread(buf);
    if (bytesRead < 0) {
        fprintf(stderr, "llread failed\n");
        exit(1);
    }

    printf("Bytes read: %d\n", bytesRead);

    if (processReceivedPacket(buf, bytesRead, filename) < 0) {
        fprintf(stderr, "Error parsing packet\n");
        exit(1);
    }
    memset(buf,0,MAX_PAYLOAD_SIZE - 6);

    while (buf[0] != 3) {
        memset(buf,0,MAX_PAYLOAD_SIZE - 6);
        int bytesRead = llread(buf);
        if (bytesRead < 0) {
            fprintf(stderr, "llread failed\n");
            memset(buf,0,MAX_PAYLOAD_SIZE - 6);
            continue;
        }

        printf("Bytes read: %d\n", bytesRead);

        if (processReceivedPacket(buf, bytesRead, filename) < 0) {
            fprintf(stderr, "Error parsing packet\n");
            exit(1);
        }
    }

}

else {
    perror("Unidentified Role\n");
    exit(-1);
}
```

```
    if (llclose(0) < 0) {
        perror("Error closing the connection\n");
        exit(-1);
    }
}
```

Link_layer.c

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

struct termios oldtioT;
int fd;
static uint8_t sequence_number = 0;

////////////////////////////////////////////////
/// AUX                                  ///
////////////////////////////////////////////////
int read_message(int fd, uint8_t *buf, int buf_size, command response) {
    int bytesRead = 0;
    reset_state();
    set_command(response);

    while (get_curr_state() != FINAL_STATE && !get_alarm_flag() && bytesRead < buf_size) {
        int bytes = read(fd, buf + bytesRead, 1);

        if (bytes == -1) {
            perror("Error reading from file descriptor");
            return -1;
        } else if (bytes == 0) {
            printf("No more data available to read.\n");
            return -1;
        }

        update_state(buf[bytesRead]);
        bytesRead++;
    }

    if (get_curr_state() != FINAL_STATE) {
        if (get_alarm_flag()) {
            printf("Failed to get response: Timeout occurred.\n");
        } else {
            printf("Failed to get response: Buffer limit reached before final state.\n");
        }
        return -1;
    }

    return bytesRead;
}

int send_message(int fd, uint8_t *frame, int msg_size, command response)
{
```

```
int bytesWritten = 0;

if (response == NO_RESPONSE) {
   bytesWritten = write(fd, frame, msg_size);
   if (bytesWritten == -1) {
      perror("Write failed");
      return -1;
   }
   return bytesWritten;
}

// WRITE RESPONSE
set_command(response);
reset_alarm_count();
reset_state();

while (get_alarm_count() < MAX_RETRIES && get_curr_state() != FINAL_STATE) {
   set_alarm_flag(FALSE);
   bytesWritten = write(fd, frame, msg_size);

   if (bytesWritten == -1) {
      perror("Write failed");
      return -1;
   }

   printf("Message sent\n");

   unsigned char buf[MAX_BUFFER_SIZE] = {0};
   alarm(ALARM_TIMEOUT);
   int bytesRead = 0;

   // LOOP UNTIL FINAL STATE OR TIMEOUT
   while (get_curr_state() != FINAL_STATE && !get_alarm_flag()) {
      if (bytesRead >= MAX_BUFFER_SIZE) {
         printf("Buffer overflow\n");
         break;
      }

      int read_byte = read(fd, buf + bytesRead, 1);
      if (read_byte == -1) {
         perror("Read error");
         return -1;
      } else if (read_byte > 0) {
         update_state(buf[bytesRead]);
         bytesRead++;
      }
   }
}

// CHECK IF FINAL STATE REACHED
if (get_curr_state() != FINAL_STATE) {
   printf("Failed to receive expected response!\n");
   return -1;
}
```

```
    return bytesWritten;
}

uint8_t *create_s_frame_buffer(uint8_t address, uint8_t control) {
    uint8_t *buffer = (uint8_t *)malloc(5);
    if (buffer == NULL) {
        return NULL;
    }
    memset(buffer, 0, 5);
    buffer[0] = FLAG;
    buffer[1] = address;
    buffer[2] = control;
    buffer[3] = BCC(buffer[1], buffer[2]);
    buffer[4] = FLAG;
    return buffer;
}

int send_s_frame(int fd, uint8_t address, uint8_t control, command response) {
    uint8_t *buffer = create_s_frame_buffer(address, control);
    if (buffer == NULL) {
        return -1;
    }
    int bytes = send_message(fd, buffer, 5, response);
    free(buffer);
    return bytes;
}

////////////////////////////////////////////////
/// LLOPEN                                   ///
////////////////////////////////////////////////
int start_transmissor(int fd){
    return send_s_frame(fd, ADDR, 0x03, RESPONSE_UA);
}

int start_receiver(int fd){
    unsigned char message[5];
    if (read_message(fd, message, 5, COMMAND_SET) < 0) return -1;
    return send_s_frame(fd, ADDR, 0x07, NO_RESPONSE);
}

int llopen(LinkLayer connectionParameters) {
    fd = open(connectionParameters.serialPort, O_RDWR | O_NOCTTY);
    if (fd < 0) {
        printf("%s", connectionParameters.serialPort);
        perror(connectionParameters.serialPort);
        exit(-1);
    }

    struct termios newtio;

    // SAVE PORT SETTINGS
    if (tcgetattr(fd, &oldtioT) == -1)  {
        perror("tcgetattr");
```

```
    exit(-1);
}

// NEW PORT SETTINGS
memset(&newtio, 0, sizeof(newtio));

newtio.c_cflag = connectionParameters.baudRate | CS8 | CLOCAL | CREAD;
newtio.c_iflag = IGNPAR;
newtio.c_oflag = 0;

// SET INPUT
newtio.c_lflag = 0;
newtio.c_cc[VTIME] = 0;
switch (connectionParameters.role) {
case TRANSMITTER:
    newtio.c_cc[VMIN] = 0;
    newtio.c_cc[VTIME] = connectionParameters.timeout;
    break;
case RECEIVER:
    newtio.c_cc[VMIN] = 1;
    break;
}

tcflush(fd, TCIOFLUSH);

// SET NEW PORT SETTINGS
if (tcsetattr(fd, TCSANOW, &newtio) == -1) {
    perror("tcsetattr");
    exit(-1);
}

printf("Start program. Please wait...\n");

switch (connectionParameters.role) {
case TRANSMITTER:
    set_role(TRANSMITTER);
    (void)signal(SIGALRM, alarm_handler);
    if (start_transmissor(fd) < 0) {
        printf("Could not start TRANSMITTER\n");
        return -1;
    }
    break;

case RECEIVER:
    set_role(RECEIVER);
    (void)signal(SIGALRM, alarm_handler);
    if (start_receiver(fd) < 0) {
        printf("Could not start RECEIVER\n");
        return -1;
    }
    break;
}

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

```
    printf("Connection Successfully\n");
    printf("-------------------------------\n");

    return fd;
}

////////////////////////////////////////////
// LLWRITE                            ///
////////////////////////////////////////////
int stuff_message(uint8_t *buffer, int start, int msg_size, uint8_t *stuffed_msg)
{
    int i = 0;

    // COPY HEADER
    for (int j = 0; j < start; ++j, ++i)
        stuffed_msg[i] = buffer[j];

    // STUFFING
    for (int j = start; j < msg_size; ++j)
    {
        if (buffer[j] == FLAG || buffer[j] == ESCAPE) {
            stuffed_msg[i++] = ESCAPE;
            stuffed_msg[i++] = buffer[j] ^ 0x20;
        } else {
            stuffed_msg[i++] = buffer[j];
        }
    }

    return i;
}

int create_stuffed_message(uint8_t *buffer, int msg_len, uint8_t *stuffed_msg) {
    msg_len = stuff_message(buffer, 4, msg_len, stuffed_msg);
    stuffed_msg[msg_len++] = FLAG;
    return msg_len;
}

uint8_t *create_i_frame_buffer(const uint8_t *data, int data_len, int packet) {
    int msg_len = data_len + 5; // Header (4 bytes) + BCC2 (1 byte)
    uint8_t *buffer = (uint8_t *)malloc(msg_len);
    if (buffer == NULL) {
        perror("Memory allocation failed");
        exit(-1);
    }

    // INIT HEADER
    buffer[0] = FLAG;
    buffer[1] = EMITTER_ADDR;
    buffer[2] = (packet << 6);
    buffer[3] = BCC(buffer[1], buffer[2]);

    // COMPUTE BCC2
    uint8_t bcc2 = 0;
    for (int i = 0; i < data_len; i++) {
```

```
        buffer[i + 4] = data[i];
        bcc2 ^= data[i];
    }

    buffer[data_len + 4] = bcc2;
    return buffer;
}

int transmit_information_frame(int fd, const uint8_t *data, int data_len, int packet) {
    uint8_t *buffer = create_i_frame_buffer(data, data_len, packet);
    if (!buffer) {
        return -1;
    }

    int msg_len = data_len + 5;
    uint8_t stuffed_msg[msg_len * 2];

    // CREATE STUFFED MESSAGE
    msg_len = create_stuffed_message(buffer, msg_len, stuffed_msg);
    free(buffer); // Free the allocated I-frame buffer after stuffing

    // SEND PROCESS
    for (int w = 0; w < 3; w++) {
        int bytes = send_message(fd, stuffed_msg, msg_len, RESPONSE_REJ);
        if (bytes == -1) {
            printf("Failed to send message correctly\n");
            return -1;
        }

        // CHECK RESPONSE
        if ((packet == 0 && get_prev_response() == PA_F1) ||
            (packet == 1 && get_prev_response() == PA_F0)) {
            printf("Positive Acknowledgement :)\n");
            return bytes; // Successful transmission
        }
        if ((packet == 0 && get_prev_response() == REJ_F1) ||
            (packet == 1 && get_prev_response() == REJ_F0)) {
            printf("Invalid message sent and rejected\n");
            continue; // Retry sending
        }
    }

    return -1;
}

int llwrite(const unsigned char *buf, int bufSize)
{
    int bytes = transmit_information_frame(fd, buf, bufSize, sequence_number);
    if (bytes == -1) {
        perror("llwrite: Error sending I-frame");
        exit(-1);
    }

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

```
    printf("Bytes successfully written: %d\n", bytes);

    sequence_number ^= 0x01;
    return bytes;
}

////////////////////////////////////////////
// LLREAD                             ///
////////////////////////////////////////////
int destuff_message(uint8_t *buffer, int start, int msg_size, uint8_t *destuffed_msg)
{
    int i = 0;
    for (int j = 0; j < start; ++j, ++i) {
        destuffed_msg[i] = buffer[j];
    }
    for (int j = start; j < msg_size; j++) {
        if (buffer[j] == ESCAPE) {
            destuffed_msg[i] = buffer[j + 1] ^ 0x20;
            j++;
            i++;
        } else {
            destuffed_msg[i] = buffer[j];
            i++;
        }
    }
    return i;
}

int llread(unsigned char *packet) {
    // ALLOC BUFFER - MESSAGE
    unsigned char *message_buffer = (unsigned char *)malloc(MAX_PAYLOAD_SIZE * 2);
    if (message_buffer == NULL) {
        perror("Memory allocation for message buffer failed");
        return -1;
    }

    // READ MESSAGE
    int bytes_read = read_message(fd, message_buffer, MAX_PAYLOAD_SIZE * 2, COMMAND_DATA);
    if (bytes_read < 0) {
        free(message_buffer);
        return -1;
    }

    // ALLOC BUFFER - DESTUFFED MESSAGE
    uint8_t *destuffed_message = (uint8_t *)malloc(MAX_PAYLOAD_SIZE);
    if (destuffed_message == NULL) {
        perror("Memory allocation for destuffed message failed");
        free(message_buffer);
        return -1;
    }

    // DESTUFFED MESSAGE
    int destuffed_message_size = destuff_message(message_buffer, 4, bytes_read, destuffed_message);
    free(message_buffer);
```

```
    if (destuffed_message_size < 0) {
        free(destuffed_message);
        fprintf(stderr, "Message destuffing failed\n");
        return -1;
    }

     // EXTRACT  BCC2
    unsigned char received_bcc2 = destuffed_message[destuffed_message_size - 2];

    // GENERATE BCC2
    unsigned char expected_bcc2 = destuffed_message[4]; // Initialize with the first byte of data
    for (int i = 5; i < destuffed_message_size - 2; ++i) {
        expected_bcc2 ^= destuffed_message[i];
    }

    // CHECK IF BCC2 RECEIVED AND BCC2 EXPECTED EQUA
    if (received_bcc2 == expected_bcc2) {

        // VALIDATE SEQUENCE NUMBER
        if ((get_control() == 0x00 && sequence_number == 0) || (get_control() == 0x40 && sequence_number == 1)) {
            send_s_frame(fd, ADDR, 0x05 | ((sequence_number ^ 0x01) << 7), NO_RESPONSE);
            memcpy(packet, destuffed_message + 4, destuffed_message_size - 6);
            free(destuffed_message);
            sequence_number ^= 0x01; // TOGGLE SEQUENCE NUMBER
            return destuffed_message_size - 6; // RETURN SIZE OF DATA
        } else {
            // DUPLICATE DATA PACKET RECEIVED
            send_s_frame(fd, ADDR, 0x05 | (sequence_number << 7), NO_RESPONSE);
            free(destuffed_message);
            return -1;
        }
    }

    // BCC2 MISMATCH HANDLING
    send_s_frame(fd, ADDR, 0x01 | ((sequence_number ^ 0x01) << 7), NO_RESPONSE);
    free(destuffed_message);
    return -1;
}

////////////////////////////////////////////////
// LLCLOSE
////////////////////////////////////////////////
int close_receiver(int fd) {
    printf("----------------------------------\n");
    printf("Disconnecting receiver...\n");
    unsigned char message[5];

    if (read_message(fd, message, sizeof(message), COMMAND_DISC) < 0) {
        fprintf(stderr, "Error reading disconnect message from receiver.\n");
        return -1;
    }

    if (send_s_frame(fd, ADDR, 0x0B, RESPONSE_UA) < 0) {
```

```
        fprintf(stderr, "Error sending UA response from receiver.\n");
        return -1;
    }

    return 0;
}

int close_transmitter(int fd) {
    printf("Disconnecting transmitter...\n");

    // Send a disconnect command
    if (send_s_frame(fd, ADDR, 0x0B, COMMAND_DISC) < 0) {
        fprintf(stderr, "Error sending disconnect command from transmitter.\n");
        return -1;
    }

    // Send an Unnumbered Acknowledge (UA) response
    if (send_s_frame(fd, ADDR, 0x07, NO_RESPONSE) < 0) {
        fprintf(stderr, "Error sending UA response from transmitter.\n");
        return -1;
    }

    return 0; // Indicate success
}

int llclose(int showStatistics) {
    int result = 0;

    switch (get_curr_role())
    {
        case TRANSMITTER:
            result = close_transmitter(fd);
            if (result < 0) {
                fprintf(stderr, "Failed to close TRANSMITTER.\n");
                return -1;
            }
            break;

        case RECEIVER:
            result = close_receiver(fd);
            if (result < 0){
                fprintf(stderr, "Failed to close RECEIVER.\n");
                return -1;
            }
            break;
        default:
            fprintf(stderr, "Unknown role during close operation.\n");
            return -1;
    }

    printf("Closing connection...\n");
    printf("----------------------------------\n");
    sleep(1);
```

```
    if (tcsetattr(fd, TCSANOW, &oldtioT) == -1)
    {
        perror("Failed to restore port settings");
        exit(-1);
    }

    close(fd);
    return 1;
}
```

Serial_port.c

```
// Serial port interface implementation
// DO NOT CHANGE THIS FILE

#include "serial_port.h"

#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <termios.h>
#include <unistd.h>

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

int spfd = -1;          // File descriptor for open serial port
struct termios oldtio; // Serial port settings to restore on closing

// Open and configure the serial port.
// Returns -1 on error.
int openSerialPort(const char *serialPort, int baudRate)
{
    // Open with O_NONBLOCK to avoid hanging when CLOCAL
    // is not yet set on the serial port (changed later)
    int oflags = O_RDWR | O_NOCTTY | O_NONBLOCK;
    spfd = open(serialPort, oflags);
    if (spfd < 0)
    {
        perror(serialPort);
        return -1;
    }

    // Save current port settings
    if (tcgetattr(spfd, &oldtio) == -1)
    {
        perror("tcgetattr");
        return -1;
    }

    // Convert baud rate to appropriate flag
```

```
  tcflag_t br;
  switch (baudRate)
  {
  case 1200:
     br = B1200;
     break;
  case 1800:
     br = B1800;
     break;
  case 2400:
     br = B2400;
     break;
  case 4800:
     br = B4800;
     break;
  case 9600:
     br = B9600;
     break;
  case 19200:
     br = B19200;
     break;
  case 38400:
     br = B38400;
     break;
  case 57600:
     br = B57600;
     break;
  case 115200:
     br = B115200;
     break;
  default:
     fprintf(stderr, "Unsupported baud rate (must be one of 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600,
115200)\n");
     return -1;
  }

  // New port settings
  struct termios newtio;
  memset(&newtio, 0, sizeof(newtio));

  newtio.c_cflag = br | CS8 | CLOCAL | CREAD;
  newtio.c_iflag = IGNPAR;
  newtio.c_oflag = 0;

  // Set input mode (non-canonical, no echo,...)
  newtio.c_lflag = 0;
  newtio.c_cc[VTIME] = 0; // Block reading
  newtio.c_cc[VMIN] = 1;  // Byte by byte

  tcflush(spfd, TCIOFLUSH);

  // Set new port settings
  if (tcsetattr(spfd, TCSANOW, &newtio) == -1)
  {
```

```
      perror("tcsetattr");
      close(spfd);
      return -1;
   }

   // Clear O_NONBLOCK flag to ensure blocking reads
   oflags ^= O_NONBLOCK;
   if (fcntl(spfd, F_SETFL, oflags) == -1)
   {
      perror("fcntl");
      close(spfd);
      return -1;
   }

   // Done
   return spfd;
}

// Restore original port settings and close the serial port.
// Returns -1 on error.
int closeSerialPort()
{
   // Restore the old port settings
   if (tcsetattr(spfd, TCSANOW, &oldtio) == -1)
   {
      perror("tcsetattr");
      return -1;
   }

   return close(spfd);
}

// Wait for a byte received from the serial port and read it (must
// check whether a byte was actually received from the return value).
// Returns -1 on error, 0 if no byte was received, 1 if a byte was received.
int readByteSerialPort(unsigned char *byte)
{
   return read(spfd, byte, 1);
}

// Write up to numBytes to the serial port (must check how many were actually
// written in the return value).
// Returns -1 on error, otherwise the number of bytes written.
int writeBytesSerialPort(const unsigned char *bytes, int numBytes)
{
   return write(spfd, bytes, numBytes);
}
```

State.c

```
#include "state.h"
```

```
state_machine state_m;

////////////////////////////////////////////////
// GETTERS
////////////////////////////////////////////////
unsigned char get_address() {
   return state_m.address;
}

unsigned char get_control() {
   return state_m.control;
}

state get_curr_state() {
   return state_m.curr_state;
}

role get_curr_role() {
   return state_m.curr_role;
}

command get_curr_command() {
   return state_m.curr_command;
}

response get_prev_response() {
   return state_m.prev_response;
}

////////////////////////////////////////////////
// SETTERS
////////////////////////////////////////////////
void set_address(unsigned char s) {
   state_m.address = s;
}

void set_control(unsigned char c) {
   state_m.control = c;
}

void set_state(state s) {
   state_m.curr_state = s;
}

void set_role(role r) {
   state_m.curr_role = r;
}

void set_command(command c) {
   state_m.curr_command = c;
}

void set_response(response r) {
   state_m.prev_response = r;
```

```
}

////////////////////////////////////////////////
// MAIN FUNCTIONS
////////////////////////////////////////////////
void update_state(unsigned char byte) {
   switch (state_m.curr_state) {
   case START_STATE:
      if (byte == FLAG)
         set_state(FLAG_RCV);
      break;

   case FLAG_RCV:
      if (byte == FLAG)
         break;

      else if (byte == ADDR) {
         set_address(byte);
         set_state(ADDRESS_RCV);
      }

      else{
         set_state(START_STATE);
      }

      break;

   case ADDRESS_RCV:
      if (byte == FLAG)
         set_state(FLAG_RCV);

      else if (byte == 0x03 && get_curr_command() == COMMAND_SET) {
         set_control(byte);
         set_state(CONTROL_RCV);
      }

      else if (byte == 0x07 && get_curr_command() == RESPONSE_UA) {
         set_control(byte);
         set_state(CONTROL_RCV);
      }

      else if (byte == 0x0B && get_curr_command() == COMMAND_DISC) {
         set_control(byte);
         set_state(CONTROL_RCV);
      }

      else if ((byte == 0x00 || byte == 0x40) && get_curr_command() == COMMAND_DATA) {
         set_control(byte);
         set_state(CONTROL_RCV);
      }

      else if ((byte == 0x05 || byte == 0x85 || byte == 0x01 || byte == 0x81) && get_curr_command() ==
RESPONSE_REJ) {
         set_control(byte);
```

```
      set_state(CONTROL_RCV);

      switch (byte) {
      case 0x05:
         set_response(PA_F0);
         break;

      case 0x85:
         set_response(PA_F1);
         break;

      case 0x01:
         set_response(REJ_F0);
         break;

      case 0x81:
         set_response(REJ_F1);
         break;
      }
   }

   else {
      set_state(START_STATE);
   }
   break;

case CONTROL_RCV:
   if (byte == (state_m.address ^ state_m.control)) {
      if (get_curr_command() == COMMAND_DATA)
         set_state(BCC_VER);

      else
         set_state(DATA_RCV);
   }

   else if (byte == FLAG)
      set_state(FLAG_RCV);

   else
      set_state(START_STATE);
   break;

case DATA_RCV:
   if (byte == FLAG)
      set_state(FINAL_STATE);

   else
      set_state(START_STATE);
   break;

case BCC_VER:
   if (byte == FLAG)
      set_state(FINAL_STATE);
   break;
```

```
    case FINAL_STATE:
      break;
    }
}

void reset_state() {
    state_m.curr_state = START_STATE;
    state_m.prev_response = RESPONSE_NULL;
}
```