

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](mailto: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:

Uma imagem com texto, captura de ecrã, Tipo de letra, documento

Descrição gerada automaticamenteUma imagem com texto, captura de ecrã, número, documento

Descrição gerada automaticamente 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.

* **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:*

Uma imagem com texto, captura de ecrã, Tipo de letra

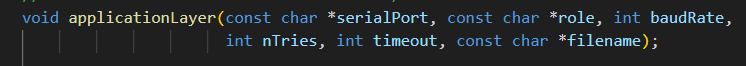
Descrição gerada automaticamente Two auxiliary data structures were used in this layer: LinkLayer, where the parameters associated with data transfer are characterized, and LinkLayerRole, which identifies whether the computer is a transmitter or receiver

Uma imagem com texto, captura de ecrã, Tipo de letra

Descrição gerada automaticamente

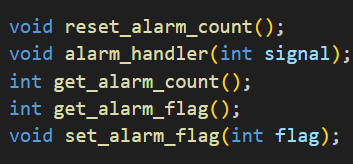
There were the functions implemented

*ApplicationLayer*

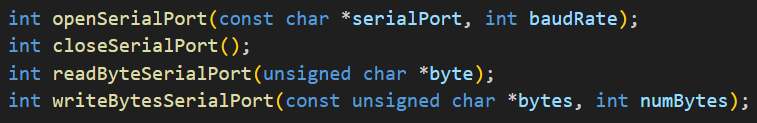


In implementing this layer, there was no need to create auxiliary data structures

*Alarm*



These were the functions implemented

*Serial Port*

There were the functions implemented

Uma imagem com texto, captura de ecrã, Tipo de letra

Descrição gerada automaticamenteUma imagem com texto, captura de ecrã, Tipo de letra

Descrição gerada automaticamenteUma imagem com texto, captura de ecrã, Tipo de letra, design

Descrição gerada automaticamenteUma imagem com texto, captura de ecrã, Tipo de letra, design

Descrição gerada automaticamente*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.

1. **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.

1. **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.

1. **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.

1. **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;

}