1st Laboratory Work

Bachelor in Informatics and Computing Engineering

Computer Networks

Diana Nunes- up202208247
Teresa Mascarenhas- up202206828

Content

1. Summary 3
2. Introduction 3
3. Architecture 3
4. Code Structure 4
4.1 Link Layer 4
4.2 Serial Port 4
4.3 Application Layer 4
5. Key Use Cases 4
6. Link Layer Protocol 5
7. Application Protocol 7
7.1 Functions Implemented in the Application Layer 7
8. Data Link Protocol Efficiency and System Validation 7
8.1 Results 8
9. Conclusion 8
10. Attachments 9
10.1 application_layer.c 9
10.2 link_layer.c 13

1. Summary

This report is a complement to the first practical assignment for the Computer Networks course, with the primary objective of data transfer. We developed an application that enables file exchange between computers over a serial port. Upon completion, we achieved all set goals, resulting in an efficient and functional application capable of file transfers without data integrity loss.

2. Introduction

This report aims to present the theoretical principles underlying the project implementation, not covered in the practical demonstration. The primary objective of this first lab assignment is to implement a data communication protocol to ensure reliable information exchange between two systems connected through a serial port.

The project required developing functions for frame creation and synchronization, connection establishment and termination, frame numbering, and confirmation of frame reception without errors and in the correct order, as well as error and flow control.

The report is organized as follows:

- **Architecture:** Overview of functional modules and interfaces.
- **Code Structure:** Overview of APIs, main data structures, functions, and their relation to the architecture.
- **Key Use Cases:** Identification of relevant use cases and function call sequences.
- **Link Layer Protocol:** Explanation of the primary functional aspects and the approach for implementation.
- **Application Protocol:** Explanation of main functional aspects and the strategy used for implementation.
- **Validation:** Description of tests conducted, with quantitative demonstration of results.
- **Protocol Efficiency:** Statistical analysis based on measurements in the developed code.
- **Conclusion:** Summary of content and learning objectives achieved.

3. Architecture

The project is structured into independent layers. The link layer includes a protocol that manages connection setup and error-controlled data transmission using techniques like byte

stuffing/destuffing. The application layer handles file reading/writing and frame sending/receiving. These layers interact directly, with the application layer depending on the link layer.

4. Code Structure

4.1 Link Layer

The link layer, developed in *link_layer.c* and *link_layer.h*, serves as the logical interface between the serial port and the application layer, managing inter-system communication with functions for configuration, connection setup/termination, data reading/writing, and error control, including packet *stuffing* and *destuffing*.

The link layer is organized with data structures for clear, adaptable configuration:

```
typedef struct
{
    char serialPort[50];
    LinkLayerRole role;         /* Transmitter (LlTx) or receiver (LlRx) */
    int baudRate;                /* Connection capacity */
    int nRetransmissions;        /* Maximum number of retries */
    int timeout;                 /* Time to timeout e . g . 1 second */
} LinkLayer;
```

4.2 Serial Port

serial_port.c and *serial_port.h* include functions essential for serial port manipulation, allowing for serial port opening and configuration, original setting restoration, and closure. Key functions include:

- *openSerialPort*: Opens and configures the serial port, returning -1 on error.
- *closeSerialPort*: Restores original settings and closes the port, returning -1 on error.
- *readByte*: Waits for and reads a byte from the serial port, returning -1 on error, 0 if no byte is received, or 1 if successful.
- *writeBytes*: Writes up to a specified number of bytes to the serial port, verifying how many were written, returning -1 on error or the count of successfully written bytes.

4.3 Application Layer

The application layer, in *application_layer.c* and *application_layer.h*, utilizes link layer logic to transfer files according to user parameters.

5. Key Use Cases

The program requires only one parameter, the serial port device (e.g., */dev/ttyS0*). In transmitter mode, it automatically locates and begins sending "pinguim.gif" if available. In receiver mode, the application waits for the transmitter to initiate the connection.

6. Link Layer Protocol

LLOPEN

```
int llopen(LinkLayer connectionParameters);
```

The **llopen** function is responsible for establishing the connection between the sender and the receiver. The function performs different operations depending on the role defined in *connectionParameters*.

- In the case of a sender, the function sends a SET control frame and activates a timer. If the expected response (UA) is not received within a set time, the SET frame will be resent, respecting a maximum limit of retransmissions as stipulated in *connectionParameters*.
- On the receiver side, the function waits for a SET frame, and upon receiving it, responds with a UA frame. The sendSupFrame function is used to send both UA and SET, constructing and transmitting the appropriate supervision frame.
- The *SetUaStateMachine* function is responsible for managing the state machine, as it checks whether the received frame matches the expected one, allowing the transition between different states.

LLWRITE

```
int llwrite(const unsigned char *buf, int bufSize);
```

The **llwrite** function is responsible for sending data organized into frames, as well as performing stuffing to ensure data integrity.

- The function initiates the construction of a data packet, which includes the Link Layer Protocol header, with BCC2 calculated using the *calculateBCC2* function.

- Next, the function performs stuffing of the message and BCC2 using the *byteStuffingTechnique*, ensuring that special characters (such as FLAG and ESC) are correctly handled.
- After assembling the frame, the function attempts to send it. A timer is activated, and the function waits for a response from the receiver (RR or REJ). If an REJ is received, the message is resent. Response verification is handled through the *infoFrameStateMachine* function.
- The retransmission process continues until the message is confirmed as received or until the maximum allowed attempts are reached.

LLREAD

```
int llread(unsigned char *packet);
```

The **llread** function is responsible for receiving data and destuffing the frames.

- Reading is performed sequentially, byte by byte. The function waits for the arrival of a data frame, and upon receiving it, verifies the validity of BCC2 at its end.
- If BCC2 is correct, a confirmation signal (RR) is sent back using the *sendSupFrame* function. Otherwise, a rejection signal (REJ) is sent, and the reading process restarts.
- The function also checks if the frame sequence number is as expected, which is crucial to avoid receiving duplicate packets.
- During the reading process, the function supervises the machine's state to ensure that the received bytes are processed correctly, while simultaneously performing destuffing of special bytes.

LLCLOSE

```
int llclose(int showStatistics);
```

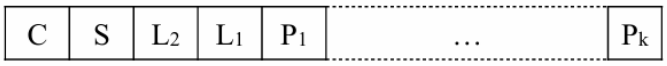
The **llclose** function terminates the connection between the sender and the receiver.

- On the sender side, the function initiates the transmission of a DISC supervision frame, waiting for the receiver's confirmation through a DISC response, indicating that the connection has been closed. This procedure is managed by a state machine, functioning similarly to the one used in the *llopen* function, with the *SetUaStateMachine* function responsible for coordinating state changes.
- After receiving the DISC response, the sender transmits a UA frame to validate the closure of the connection.
- On the receiver side, the function waits for the arrival of a DISC. As soon as it receives the DISC, it responds with the same frame and then waits for the UA frame to confirm

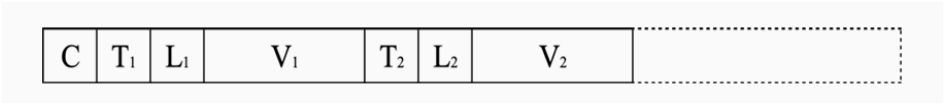
that the connection has been closed. If DISC is not received within the maximum number of attempts, the function indicates a failure in closing the connection.

7. Application Protocol

7.1 Functions Implemented in the Application Layer

- **Control Packet Creation:** *createCtrlPacket* generates control packets containing file status, name, and size.
- 

- **Data Packet Creation:** *createDataPacket* divides file content into data packets with sequence numbers and fragment size.



- **Control Packet Parsing:** *parseCtrlPacket* processes received control packets, extracting file name and validating information.
- **Checksum Calculation and Verification:** *calculateChecksum* and *verifyChecksum* ensure data integrity by detecting transmission errors.
- **Application Layer Management:** *applicationLayer* handles data sending/receiving, connection management, timing, and efficiency.

8. Data Link Protocol Efficiency and System Validation

To validate the developed data link protocol's efficiency, tests were conducted across varying transmission rates and packet sizes, comparing these empirical results with the theoretical efficiency of the Stop & Wait protocol.

Test Methodology

Efficiency was assessed by varying baud rate and packet size, comparing theoretical expectations with observed performance based on transmission time, propagation, and processing delays.

Theoretical vs. Observed Efficiency

The Stop & Wait protocol, which inherently includes idle time between frames, limits efficiency. Its theoretical efficiency is defined by:

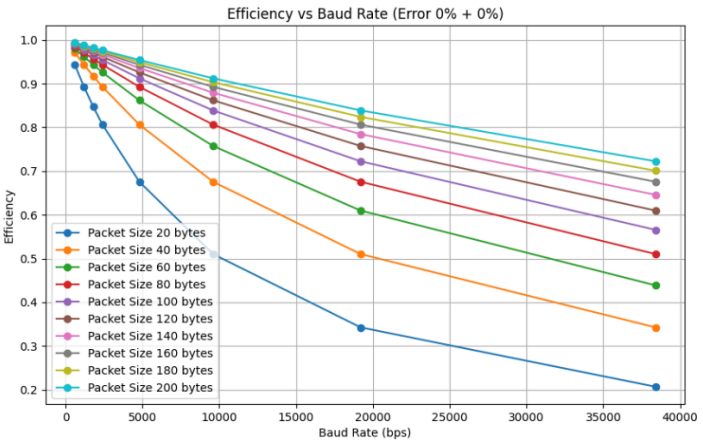
$$Efficiency = Tframe \div (Tframe + 2 \cdot Tprop)$$

where $Tframe$ is frame transmission time, and $Tprop$ is propagation delay. In our tests, we noted close alignment with this model, particularly with lower transmission rates and smaller packets.

8.1 Results

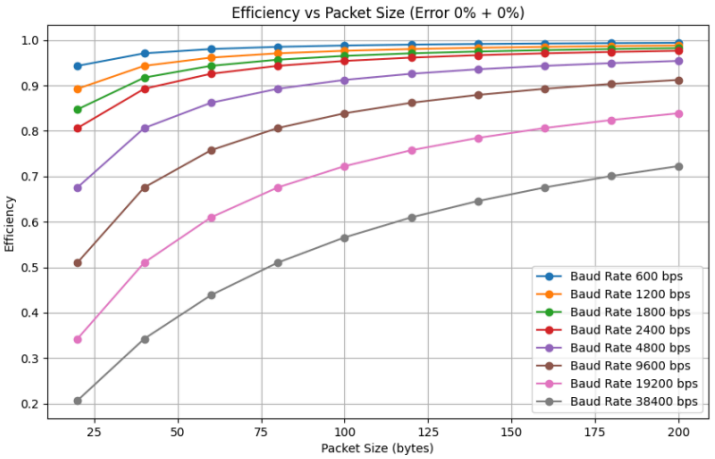
Efficiency vs. Baud Rate:

Higher transmission rates increase efficiency, with small packets benefiting most. For larger packets, the efficiency stabilized at around 90%, indicating an ideal transmission rate..



Efficiency vs. Packet Size:

Efficiency grew with packet size up to a point between 100 and 120 bytes, after which gains decreased. Very large packets reduced efficiency at lower transmission rates.



In summary, optimal efficiency depends on both transmission rate and packet size, with certain configurations providing the highest performance. The graphs highlight these variations, identifying the most efficient setups for different scenarios.

9. Conclusion

In conclusion, the primary objective was achieved. Developing the project required understanding concepts essential to layer independence. The application layer, while depending on the link layer for functionality, operates independently, focusing on how to access the link layer's service. This approach not only clarified the layered architecture but also provided

practical experience in achieving robust data communication over a serial interface.

10. Attachments

10.1 application_layer.c

```c
#include "application_layer.h"
#include "link_layer.h"
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

unsigned char* createCtrlPacket(int c, int* fileSize, const char *filename);
unsigned char* createDataPacket(int sequence, int dataSize, unsigned char* data);
unsigned char* parseCtrlPacket(unsigned char* packet, int size);
unsigned short calculateChecksum(unsigned char* data, int length);
int verifyChecksum(unsigned char* packet, int packetSize);

void applicationLayer(const char *serialPort, const char *role, int baudRate,
                      int nTries, int timeout, const char *filename)
{
    LinkLayer linkLayer;
    strcpy(linkLayer.serialPort, serialPort);
    if (strcmp(role, "tx") == 0) {
        linkLayer.role = LlTx;
    } else {
        linkLayer.role = LlRx;
    }
    linkLayer.baudRate = baudRate;
    linkLayer.nRetransmissions = nTries;
    linkLayer.timeout = timeout;

    int fd = llopen(linkLayer);
    if (fd < 0) {
        perror("Connection error\n");
        exit(-1);
    }

    switch (linkLayer.role) {
        case LlTx: {
            FILE* file = fopen(filename, "rb");
            if (file == NULL) {
                perror("File doesn't exist\n");
                exit(-1);
            }
```

```
            fseek(file, 0L, SEEK_END);
            int fileSize = ftell(file);
            fseek(file, 0L, SEEK_SET);

            unsigned char* ctrlPacket = createCtrlPacket(1, &fileSize, filename);
            int ctrlPacketSize = 5 + sizeof(fileSize) + strlen(filename);
            if (llwrite(ctrlPacket, ctrlPacketSize) == -1) {
                perror("Error in ctrlPacket \n");
                exit(-1);
            }

            int dataPacketCount = 0;
            int nDataPackets = (fileSize + MAX_PAYLOAD_SIZE - 1) /
MAX_PAYLOAD_SIZE;

            for (int i = 0; i < nDataPackets; ++i) {
                int dataSize = (i == nDataPackets - 1) ? (fileSize %
MAX_PAYLOAD_SIZE) : MAX_PAYLOAD_SIZE;
                if (dataSize == 0) dataSize = MAX_PAYLOAD_SIZE;

                unsigned char* data = (unsigned char*) malloc(dataSize);
                fread(data, sizeof(unsigned char), dataSize, file);

                unsigned char* dataPacket = createDataPacket(dataPacketCount,
dataSize, data);
                if (llwrite(dataPacket, 4 + dataSize + sizeof(unsigned short)) ==
-1) {
                    perror("Error in dataPacket \n");
                    free(data);
                    exit(-1);
                }
                dataPacketCount++;
                free(data);
            }

            unsigned char* endCtrlPacket = createCtrlPacket(3, &fileSize,
filename);
            if (llwrite(endCtrlPacket, ctrlPacketSize) == -1) {
                perror("Error in final ctrlPacket \n");
                exit(-1);
            }
            fclose(file);

            if (llclose(fd) < 0) {
                perror("Closing error\n");
                exit(-1);
            }
            break;
        }
```

```
        case LlRx: {
            unsigned char *packet = (unsigned char *)malloc(MAX_PAYLOAD_SIZE +
sizeof(unsigned short));
            if (packet == NULL) {
                perror("Failed to allocate memory for packet");
                exit(-1);
            }

            int packetSize = -1;

            while ((packetSize = llread(packet)) < 0);
            if (packetSize < 5) {
                fprintf(stderr, "Error reading control packet\n");
                free(packet);
                exit(EXIT_FAILURE);
            }

            unsigned char* fileName = parseCtrlPacket(packet, packetSize);
            if (fileName == NULL) {
                fprintf(stderr, "Failed to parse control packet\n");
                free(packet);
                exit(EXIT_FAILURE);
            }

            FILE *newFile = fopen("penguin_received.gif", "wb+");
            if (newFile == NULL) {
                perror("Error opening file for writing");
                free(packet);
                exit(EXIT_FAILURE);
            }

            while (1) {
                while ((packetSize = llread(packet)) < 0);
                if (packetSize == 0) {
                    break;
                }

                if (verifyChecksum(packet, packetSize)) {
                    if (packet[0] == 2) {
                        unsigned char *buffer = (unsigned char
*)malloc(packetSize - 4 - sizeof(unsigned short));
                        memcpy(buffer, packet + 4, packetSize - 4 -
sizeof(unsigned short));
                        fwrite(buffer, sizeof(unsigned char), packetSize - 4 -
sizeof(unsigned short), newFile);
                        free(buffer);
                    }
                } else {
                }
            }
```

```
            fclose(newFile);
            free(packet);
            break;
        }
    }
}

unsigned char* parseCtrlPacket(unsigned char* packet, int size) {
    unsigned char fileNameNBytes = packet[3 + packet[2] + 1];
    unsigned char *fileName = (unsigned char *)malloc(fileNameNBytes + 1);
    if (fileName == NULL) {
        perror("Failed to allocate memory for file name");
        return NULL;
    }

    memcpy(fileName, packet + 3 + packet[2] + 2, fileNameNBytes);
    fileName[fileNameNBytes] = '\0';

    return fileName;
}

unsigned char* createCtrlPacket(int c, int* fileSize, const char *filename) {
    int fileNameLength = strlen(filename);
    int fileSizeLength = sizeof(*fileSize);

    unsigned char* packet = (unsigned char*)malloc(5 + fileSizeLength +
fileNameLength + sizeof(unsigned short));
    if (!packet) {
        perror("Failed to allocate memory for control packet");
        return NULL;
    }

    packet[0] = c;
    int i = 1;

    packet[i++] = 0;
    packet[i++] = fileSizeLength;
    memcpy(&packet[i], fileSize, fileSizeLength);
    i += fileSizeLength;

    packet[i++] = 1;
    packet[i++] = fileNameLength;
    memcpy(&packet[i], filename, fileNameLength);
    i += fileNameLength;

    unsigned short checksum = calculateChecksum(packet, i);
    memcpy(packet + i, &checksum, sizeof(unsigned short));

    return packet;
```

```
}

unsigned char* createDataPacket(int sequence, int dataSize, unsigned char* data)
{
    unsigned char* packet = (unsigned char*)malloc(4 + dataSize + sizeof(unsigned
short));
    if (!packet) {
        perror("Failed to allocate memory for data packet");
        return NULL;
    }

    packet[0] = 2;
    packet[1] = sequence;
    packet[2] = (dataSize >> 8) & 0xFF;
    packet[3] = dataSize & 0xFF;

    memcpy(packet + 4, data, dataSize);

    unsigned short checksum = calculateChecksum(packet, 4 + dataSize);
    memcpy(packet + 4 + dataSize, &checksum, sizeof(unsigned short));

    return packet;
}

unsigned short calculateChecksum(unsigned char* data, int length) {
    unsigned short checksum = 0;
    for (int i = 0; i < length; i++) {
        checksum += data[i];
    }
    return checksum;
}

int verifyChecksum(unsigned char* packet, int packetSize) {
    unsigned short receivedChecksum;
    memcpy(&receivedChecksum, packet + packetSize - sizeof(unsigned short),
sizeof(unsigned short));
    return calculateChecksum(packet, packetSize - sizeof(unsigned short)) ==
receivedChecksum;
}
```

10.2 link_layer.c

```
#include "link_layer.h"
#include "serial_port.h"

#define _POSIX_SOURCE 1

int alarmEnabled = FALSE;
```

```c
int firstFrame = TRUE;
int alarmCount = 0;
int max_retransmissions = 0;
int timeout = 0;

unsigned char send = 0;
unsigned char receive = 1;

extern int fd;

typedef enum
{
   START,
   FLAG_RCV,
   A_RCV,
   C_RCV,
   BCC1_OK,
   BCC2_OK,
   STOP_,
   DETECTED_ESC,
   PROCESSING
} LLState;

typedef struct {
    unsigned char expected_addr;
    unsigned char send_ctrl;
    unsigned char recv_ctrl;
    unsigned char response_ctrl;
} RoleParams;

typedef struct {
    int totalTransmissions;
    int successfulTransmissions;
    int failedTransmissions;
    int totalBytesSent;
} LinkStatistics;

LinkStatistics stats = {0};

#define T_SECONDS 3
#define BUF_SIZE 5

#define FLAG    0x7E
#define A1      0x03
#define A2      0x01
#define ESC     0x7D

#define SET     0X03
#define UA      0x07
#define RR0     0xAA
```

```
#define RR1     0xAB
#define REJ0    0x54
#define REJ1    0x55
#define DISC    0x0B
#define BCC1(a,c) ((a)^(c))

#define INFO0    0x00
#define INFO1    0x80


#define RR(Nr) ((Nr == 0) ? RR0 : RR1)
#define REJ(Nr) ((Nr == 0) ? REJ0 : REJ1)
#define I(Ns)  ((Ns == 0) ? INFO0 : INFO1)


void alarmHandler(int signal);
void initAlarm();
int sendSupFrame(int fd, unsigned char addr, unsigned char ctrl);
LLState SetUaStateMachine(int fd, unsigned char expectedAddr, unsigned char
expectedCtrl, unsigned char expectedBCC);
unsigned char calculateBCC2(const unsigned char *buf, int bufSize);
void byteStuffingTechnique(unsigned char **frame, int *frameSize, unsigned char
byte, int *j);
unsigned char infoFrameStateMachine(int fd);



void alarmHandler(int signal)
{
    alarmEnabled = FALSE;
    alarmCount++;
    printf("Waiting... #%d\n", alarmCount);
}

void initAlarm() {
    struct sigaction act = { 0 };
    act.sa_handler = &alarmHandler;
    if (sigaction(SIGALRM, &act, NULL) == -1) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }
}

int sendSupFrame(int fd, unsigned char addr, unsigned char ctrl) {
    unsigned char frame[BUF_SIZE] = {FLAG, addr, ctrl, addr ^ ctrl, FLAG};
    int bytes = write(fd, frame, BUF_SIZE);
    return bytes;
}
```

```
LLState SetUaStateMachine (int fd, unsigned char expectedAddr, unsigned char
expectedCtrl, unsigned char expectedBCC) {
    unsigned char buf[BUF_SIZE] = {0};
    LLState state = START;
    while (state != STOP_){
        int bytes = read(fd, buf, BUF_SIZE);
        if (bytes < 0) {
            perror("Error reading from serial port");
            return START;
        }

        for (int i = 0; i < bytes; i++) {
            unsigned char byte = buf[i];
            switch (state) {
                case START:
                    if (byte == FLAG) {
                        state = FLAG_RCV;
                    }
                    break;

                case FLAG_RCV:
                    if (byte == FLAG) {
                        state = START;
                    } else if (byte == expectedAddr) {
                        state = A_RCV;
                    }
                    break;

                case A_RCV:
                    if (byte == FLAG) {
                        state = FLAG_RCV;
                    } else if (byte == expectedCtrl) {
                        state = C_RCV;
                    } else {
                        state = START;
                    }
                    break;

                case C_RCV:
                    if (byte == FLAG) {
                        state = FLAG_RCV;
                    } else if (byte == expectedBCC) {
                        state = BCC1_OK;
                    } else {
                        state = START;
                    }
                    break;

                case BCC1_OK:
                    if (byte == FLAG) {
```

```
                    state = STOP_;
                } else {
                    state = START;
                }
                break;

            default:
                state = START;
                break;
        }
    }
    return state;
}

int llopen(LinkLayer connectionParameters) {
    fd = openSerialPort(connectionParameters.serialPort,
connectionParameters.baudRate);
    if (fd < 0) {
        return -1;
    }

    max_retransmissions = connectionParameters.nRetransmissions;
    timeout = connectionParameters.timeout;

    switch (connectionParameters.role) {
        case LlTx: {

            while (alarmCount < max_retransmissions) {
                initAlarm();

                if (alarmEnabled == FALSE) {
                    sendSupFrame(fd, A1, SET);
                    alarm(timeout);
                    alarmEnabled = TRUE;
                }

                LLState state = SetUaStateMachine(fd, A1, UA, BCC1(A1, UA));

                if (state == STOP_) {
                    alarm(0);
                    return 1;
                }
            }

            printf("Maximum number of timeouts exceeded. Communication could not
be established.\n");
            return -1;
        }
```

```
        case LlRx: {

            LLState state = SetUaStateMachine(fd, A1, SET, BCC1(A1, SET));

            if (state == STOP_) {
                sendSupFrame(fd, A1, UA);
                return 1;
            }

            return -1;
        }
    }
    return -1;
}


unsigned char calculateBCC2(const unsigned char *buf, int bufSize) {
    unsigned char BCC2 = buf[0];
    for (int i = 1; i < bufSize; i++) {
        BCC2 ^= buf[i];
    }
    return BCC2;
}

void byteStuffingTechnique(unsigned char **frame, int *size, unsigned char byte,
int *index) {
    if (byte == FLAG || byte == ESC) {
        *frame = realloc(*frame, *size + 1);
        (*frame)[*index] = ESC;
        (*frame)[(*index) + 1] = byte ^ 0x20;
        *index += 2;
        *size += 1;
    } else {
        (*frame)[*index] = byte;
        (*index)++;
    }
}



unsigned char infoFrameStateMachine(int fd) {
    unsigned char infoFrame = 0;
    LLState state = START;

    unsigned char buf[BUF_SIZE] = {0};
    while (state != STOP_ && !alarmEnabled) {
        int bytesRead = read(fd, buf, BUF_SIZE);

        if (bytesRead <= 0) {
            continue;
```

```
        }

        for (int i = 0; i < bytesRead; i++) {
            unsigned char byte = buf[i];
            switch (state) {
                case START:
                    if (byte == FLAG) {
                        state = FLAG_RCV;
                    }
                    break;
                case FLAG_RCV:
                    if (byte == A1) {
                        state = A_RCV;
                    } else if (byte != FLAG) {
                        state = START;
                    }
                    break;
                case A_RCV:
                    if (byte == RR0 || byte == RR1 || byte == REJ0 || byte ==
REJ1 || byte == DISC) {
                        infoFrame = byte;
                        state = C_RCV;
                    } else if (byte == FLAG) {
                        state = FLAG_RCV;
                    } else {
                        state = START;
                    }
                    break;
                case C_RCV:
                    if (byte == BCC1(A1, infoFrame)) {
                        state = BCC1_OK;
                    } else if (byte == FLAG) {
                        state = FLAG_RCV;
                    } else {
                        state = START;
                    }
                    break;
                case BCC1_OK:
                    if (byte == FLAG) {
                        state = STOP_;
                    } else {
                        state = START;
                    }
                    break;
                default:
                    state = START;
                    break;
            }
        }
    }
```

```
    return infoFrame;
}

int llwrite(const unsigned char *buf, int bufSize) {
    int totalSize = 6 + bufSize;
    unsigned char *frame = (unsigned char *) malloc(totalSize);

    frame[0] = FLAG;
    frame[1] = A1;
    frame[2] = I(send);
    frame[3] = frame[1] ^ frame[2];

    memcpy(frame + 4, buf, bufSize);

    int j = 4;
    for (int i = 0; i < bufSize; i++) {
        byteStuffingTechnique(&frame, &totalSize, buf[i], &j);
    }

    unsigned char BCC2 = calculateBCC2(buf, bufSize);
    byteStuffingTechnique(&frame, &totalSize, BCC2, &j);

    frame[j++] = FLAG;

    int transmission = 0;
    int check_rej = 0, check_rr = 0;

    while (transmission < max_retransmissions) {
        alarmEnabled = FALSE;
        alarm(timeout);
        check_rej = 0;
        check_rr = 0;

        while (!alarmEnabled && !check_rej && !check_rr) {
            write(fd, frame, j);

            stats.totalBytesSent += j;

            unsigned char res = infoFrameStateMachine(fd);

            if (!res) {
                continue;
            } else if (res == REJ(0) || res == REJ(1)) {
                check_rej = 1;
            } else if (res == RR(0) || res == RR(1)) {
                check_rr = 1;
                send = (send + 1) % 2;
            } else {
                continue;
            }
```

```
        }

        if (check_rr) {
            stats.totalBytesSent += totalSize;
            alarm(0);
            alarmCount = 0;
            break;
        }

        printf("Transmission failed, attempt %d.\n", transmission + 1);
        transmission++;
    }

    free(frame);

    if (check_rr) {
        return totalSize;
    } else {
        printf("Max transmissions reached. Closing link layer.\n");
        llclose(fd);
        return -1;
    }
}


int processingData(unsigned char *packet, unsigned char byte, int *i, unsigned
char infoFrame, LLState *state) {
    unsigned char bcc2 = packet[*i - 1];
    (*i)--;

    unsigned char calculatedBCC2 = calculateBCC2(packet, *i);
    if (bcc2 == calculatedBCC2) {
        sendSupFrame(fd, A1, RR(infoFrame));
        *state = STOP_;
        return 1;
    } else {
        sendSupFrame(fd, A1, REJ(infoFrame));
        *i = 0;
        return 0;
    }
}


int llread(unsigned char *packet) {
    unsigned char byte;
    int i = 0;
    LLState state = START;
    int bytesRead = 0;
    unsigned char infoFrame;
    int stop = 0;
```

```
while (!stop && state != STOP_) {
    if (read(fd, &byte, 1) > 0) {
        bytesRead++;

        switch (state) {
            case START:
                if (byte == FLAG) {
                    state = FLAG_RCV;
                }
                break;

            case FLAG_RCV:
                if (byte == A1) {
                    state = A_RCV;
                } else if (byte != FLAG) {
                    state = START;
                }
                break;

            case A_RCV:
                if (byte == I(0) || byte == I(1)) {
                    infoFrame = byte;
                    state = C_RCV;
                } else if (byte == FLAG) {
                    state = FLAG_RCV;
                }else if (byte == SET) {
                    sendSupFrame(fd, A1, UA);
                    state = START;
                    continue;
                }
                 else if (byte == DISC) {
                    sendSupFrame(fd, A1, DISC);
                    return 0;
                }
                else state = START;
                break;

            case C_RCV:
                if (byte == (A1 ^ infoFrame)) {
                    state = PROCESSING;
                } else if (byte == FLAG) {
                    state = FLAG_RCV;
                } else {
                    state = START;
                    printf("Error on BCC1, going back to START\n");
                    return -1;
                }
                break;
```

```
                case PROCESSING:
                    if (byte == FLAG) {
                        if (processingData(packet, byte, &i, infoFrame, &state))
{
                            stop = 1;
                        }
                    } else if (byte == ESC) {
                        state = DETECTED_ESC;
                    } else {
                        packet[i++] = byte;
                    }
                    break;

                case DETECTED_ESC:
                    if (byte == ESC || byte == FLAG) {
                        packet[i++] = byte;
                    } else {
                        packet[i++] = byte ^ 0x20;
                    }
                    state = PROCESSING;
                    break;

                default:
                    break;
            }
        } else {
            printf("Read error. Sending REJ.\n");
            sendSupFrame(fd, A1, REJ(infoFrame));
            return -1;
        }
    }

    return i;
}

int llclose(int showStatistics) {
    LLState state = START;
    initAlarm();
    alarmCount = 0;
    alarmEnabled = FALSE;

    int discFrameSize = sizeof(A1) + sizeof(DISC) + sizeof(BCC1(A1, DISC));

    while (alarmCount < max_retransmissions) {
        if (!alarmEnabled) {
            sendSupFrame(fd, A1, DISC);
            alarm(timeout);
            alarmEnabled = TRUE;
```

```
            stats.totalTransmissions++;
            stats.totalBytesSent += discFrameSize;
        }

        state = SetUaStateMachine(fd, A1, DISC, BCC1(A1, DISC));
        if (state == STOP_) {
            stats.successfulTransmissions++;
            alarm(0);
            break;
        }
    }

    if (state != STOP_) {
        printf("Failed after retries.\n");
        stats.failedTransmissions++;
        return -1;
    }


    int uaFrameSize = sizeof(A1) + sizeof(UA) + sizeof(BCC1(A1, UA));
    sendSupFrame(fd, A1, UA);
    stats.totalBytesSent += uaFrameSize;

    printf("Closed with success.\n");

    if (showStatistics) {
        printf("Statistics:\n");
        printf("Total Transmissions: %d\n", stats.totalTransmissions);
        printf("Successful Transmissions: %d\n", stats.successfulTransmissions);
        printf("Failed Transmissions: %d\n", stats.failedTransmissions);
        printf("Total Bytes Sent: %d\n", stats.totalBytesSent);
    }

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