Faculdade de Engenharia da Universidade do Porto

# Data Link Protocol

1st Lab Work of RCOM

**Supervisor:**

Eduardo Almeida enalmeida@fe.up.pt

**Authors:**

Bruno Huang up202207517@fe.up.pt

Ricardo Yang up202208465@fe.up.pt

## Summary

This project was developed for the Redes e Computadores (RCOM) course, with the objective of implementing a data link layer protocol according to specific requirements.

Through this work, we gained a deeper understanding of key concepts in communication protocols, such as data transmission and reception through a serial port, the Stop-and-Wait ARQ mechanism, the stuffing mechanism, and layer independency.

## 1. Introduction

This project aims to develop a data link protocol to enable file transfer between two computers connected via an RS-232 serial cable, by implementing both transmitter and receiver functionalities. The report is organized as follows:

- Architecture: Functional blocks and interfaces.
- Code Structure: APIs, main data structures and main functions。
- Main Use Cases: Identification of core use cases and sequences of function calls.
- Logical Link Protocol: Main functionalities of the link layer and the implementation strategy used.
- Application Protocol: Main functionalities of the application layer and the implementation strategy used.
- Validation: Description of the tests performed and results.
- Data Link Protocol Efficiency: Quantitative analysis of protocol efficiency, including a theoretical assessment of the Stop & Wait protocol.
- Conclusions: Synthesis of the report’s findings and reflections on achieved learning objectives.
- Appendix 1: Efficiency Graphs.
- Appendix 2: Source Code.

# 2. Architecture

This project is structured with two main layers: Link Layer and Application Layer, ensuring their independence.

The Link Layer manages data transmission, including framing, error checking and retransmission, providing a reliable connection for data exchange. The Application Layer handles file-related tasks, structuring file data into packets and sending or receiving them in sequence. It relies on the Link Layer for secure transmission.

Additionally, the project includes supplementary modules, such as Protocol module with protocol's related macros and a Statistics module to track and monitor the performance metrics of the protocol.

# 3. Code Structure

### Serial Port

The **serial_port** code was provided, so no modifications were made. APIs:

```
#include <fcntl.h>
#include <termios.h>
#include <unistd.h>
```

### Application Layer

No specific data structures were required for implementation. The main functions:

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

int readPacketControl(unsigned char *buff, int *isEnd);
int readPacketData(unsigned char *buff, size_t *newSize, unsigned char
*dataPacket);
int sendPacketControl(unsigned char C, const char *filename, size_t
file_size);
int sendPacketData(size_t nBytes, unsigned char *data);
```

### Link Layer

The main data structures:

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

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

typedef enum { START_STATE, FLAG_RCV, A_RCV, C_RCV, BCC_OK, STOP_STATE,
DATA_STATE, ESC_STATE } LinkLayerState;
```

The main functions:

```c
int llopen(LinkLayer connectionParameters);
int llwrite(const unsigned char *buf, int bufSize);
int llread(unsigned char *packet);
int llclose(int showStatistics);
```

**Statistics**

The main data structures:

```c
typedef struct {
    unsigned int bytesRead;
    unsigned int nFrames;
    unsigned int errorFrames;
    unsigned int retransmissions;
    struct timeval startTime;
    struct timeval endTime;
} Statistics;
```

The main functions:

```c
double optimal_efficiency(int baudrate, int maxPayload);
double actual_efficiency(Statistics stats, int baudrate);
```

## 4. Main Use Cases

The program operates in two modes: transmitter (TX) and receiver (RX). First, both the transmitter and receiver use `llopen` to establish the serial port connection through an initial handshake. The transmitter then creates packets and sends them using `llwrite`, with the `sendPacketControl` and `sendPacketData` functions. Meanwhile, the receiver continually calls `llread` until it reads the end control byte, signaling the end of file transmission, and interprets the packets using `readPacketControl` and `readPacketData`. Finally, both sides call `llclose` to disconnect the serial port and show the connection statistics using `showStatisticsTerminal`.

# 5. Logical Link Protocol

The main purpose of the link layer is to manage framing, error detection, and retransmission to ensure secure and reliable data exchange. The four key functions are outlined below.

## 5.1. llopen - Connection Establishment

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

The `llopen` function takes **connectionParameters**, which includes information about the serial port setup, role, and error handling. Its main purpose is to establish a connection between the transmitter and receiver. The transmitter starts by sending a **SET** supervision frame and waits for the receiver to respond with a **UA** acknowledgment frame. If the transmitter does not receive a response within the time, it will resend the **SET** frame according to the defined number of attempts.

## 5.2. llwrite - Data Transmission

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

The `llwrite` function takes a buffer (**buf**) and its size (**bufSize**) as parameters. It is responsible for sending an information frame to the receiver. The function prepares the frame using byte stuffing to prevent data values from conflicting with **FLAG** or **ESCAPE** bytes, which could cause data loss. After sending the frame, if the receiver replies with an **RR** frame, it means the data was received successfully. If the reply is **REJ**, the transmitter will resend the frame based on the defined number of attempts.

### 5.3. llread - Data Reception

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

The `llread` function reads incoming frames and creates packets. It performs byte destuffing and checks the protection fields **BCC1** and **BCC2**. If everything checks out, it sends an **RR** frame to indicate readiness. If there is an issue, it sends a **REJ** frame to request retransmission. If a duplicate frame is received, it acknowledges it with an RR but not processes further to prevent the same frame from being processed multiple times.

### 5.4. llclose - Data Termination

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

The `llclose` function is called to terminate the connection after all packets have been sent or if the retransmission limit is reached. The transmitter first sends a **DISC** frame to request disconnection. Once the receiver acknowledges this with a **UA** frame, the connection is closed and the statistics are displayed if requested.

## 6. Application Protocol

The Application Layer handles direct interaction with the file being sent, following the initial connection setup between the transmitter and receiver.

Once the connection is established, the transmitter first sends a **start** control packet using the `sendPacketControl` function. This packet is coded as TLV (Type-Length-Value) which includes file properties such as its name and size. After the start packet is received, the transmitter begins sending **data** packets of a predefined size, containing chunks of the file's data, using `sendPacketData` function. Finally, to signal the end of transmission, the transmitter sends an **end** control packet with the same content as the start packet.

On the receiver side, `llread` is constantly called to read each packet until the end packet is received, signaling the completion of the file transmission. For each read call, the receiver identifies the packet type (control or data) and processes it using either `readPacketControl` or `readPacketData`. The file is then reconstructed "piece by piece".

In this project, the receiver sets a file name at the start to ensure that **Makefile** can verify the correctness of the transferred file. However, it’s possible to name the received file the same as the original by reading the file name from the start control packet.

# 7. Validation

We validated our program through various tests, including transmission under normal conditions, with cable noise, after cable disconnection and reconnection, with a specified frame error rate (FER), at different baud rates, retransmission counts, timeout values, payload sizes, and with duplicated frames.

All test cases confirmed the program’s functionality when using suitable parameter settings. However, given the Stop-and-Wait ARQ mechanism innate limitations, efficiency can be constrained by timeout intervals and retransmission overhead.

# 8. Data Link Protocol Efficiency

To evaluate the protocol efficiency, a `statistics` module was created into the code to record data and perform calculations with minimal impact on data transfer.

The resulting data and graphs are presented in the Appendix 1, where, in all graphs, x-axis represents the variable values and y-axis the efficiency (S).

*Default test parameters* are: baud rate of 9600, payload size of 1000 bytes, error rate of 0%, and a propagation delay of 0 ms.

## 8.1. Varying Propagation Delay

The `usleep` function from `unists.h` was added to simulate the physical delay typical of a real serial port. The results show that as delay increases, total propagation time also rises, leading to a reduction in overall efficiency.

## 8.2. Varying BCC1 Error Probability

A macro **BCC1_ERROR** was added in `statistics.h` to simulate errors in **BCC1** at various probabilities. The results indicate that the efficiency decreases significantly as the error rate rises. This effect is more evident than BCC2 errors, as a BCC1 error prevents any receiver response, causing timeouts and retransmissions from the transmitter.

### 8.3. Varying BCC2 Error Probability

A macro **BCC2_ERROR** was added in `statistics.h` to simulate errors in **BCC2**, prompting the receiver to send a **REJ** frame when an error is detected. The results indicate that efficiency decreases as the error probability increases.

### 8.4. Varying Baud Rate

During manual tests, baud rate was adjusted with different values. The results indicate that efficiency declines as baud rate increases, suggesting that lower baud rates may better utilize physical link characteristics, potentially due to reduced error rates and fewer retransmissions required at lower speeds.

### 8.5. Varying Payload size

The **MAX_PAYLOAD_SIZE** macro in `link_layer.h` was modified to test different payload sizes. Efficiency increased slightly with larger payload sizes. However, in real scenarios, larger frames could lead to more bad frames, which would decrease efficiency due to more frequent retransmissions.

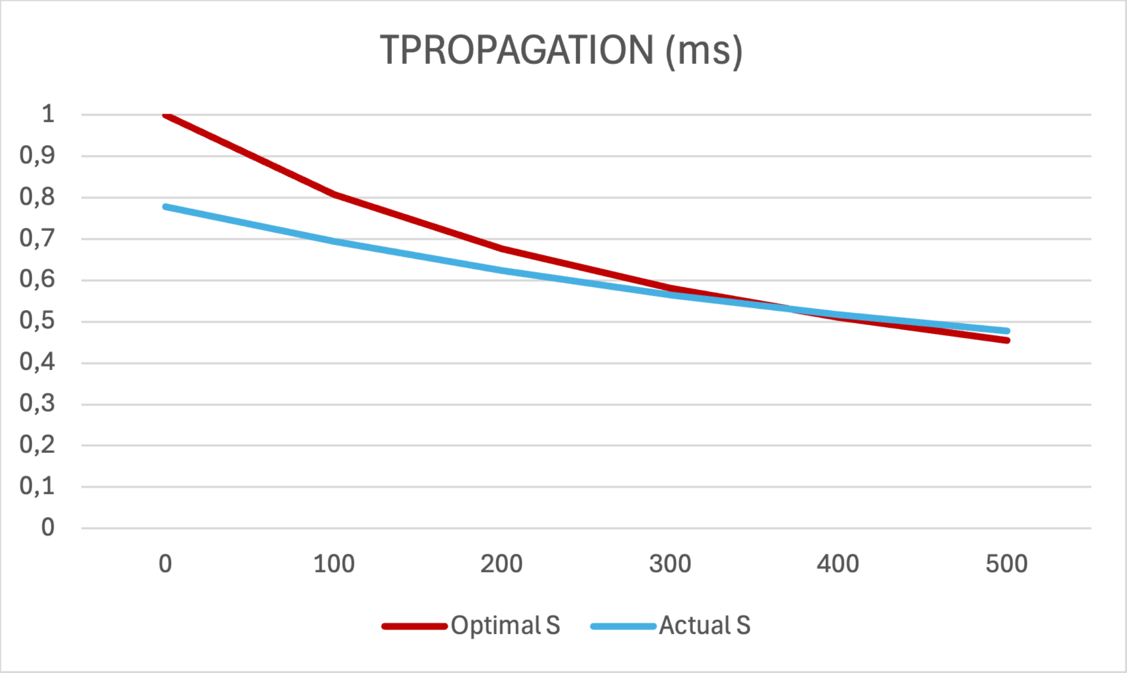
## 9. Conclusions

This project was a great opportunity to apply data transfer concepts in a practical setting. We observed that while the Stop-and-Wait ARQ protocol works for reliable data transfer, its efficiency in practice is significantly lower than expected.

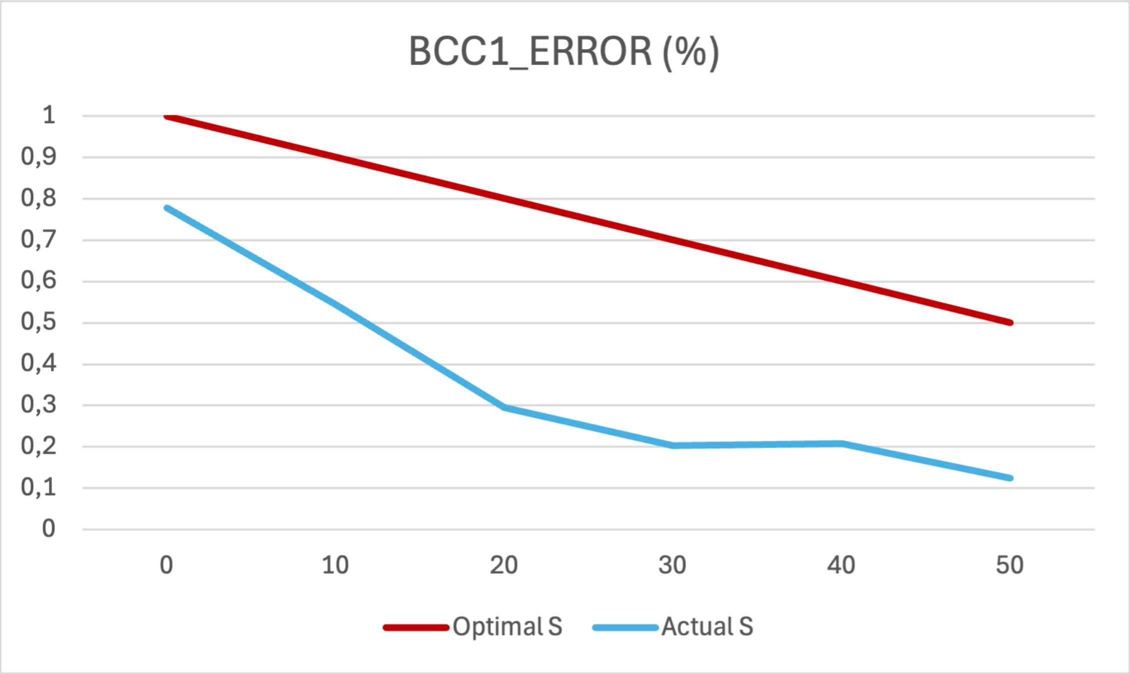
Through this work, we gained experience with key mechanisms like byte stuffing, framing, and packet handling, which are essential for maintaining data integrity. Overall, this project helped us understand the strengths and limitations of basic data transfer protocols and gave us practical insight into network protocol design.

# Appendix 1 - Efficiency Graphs

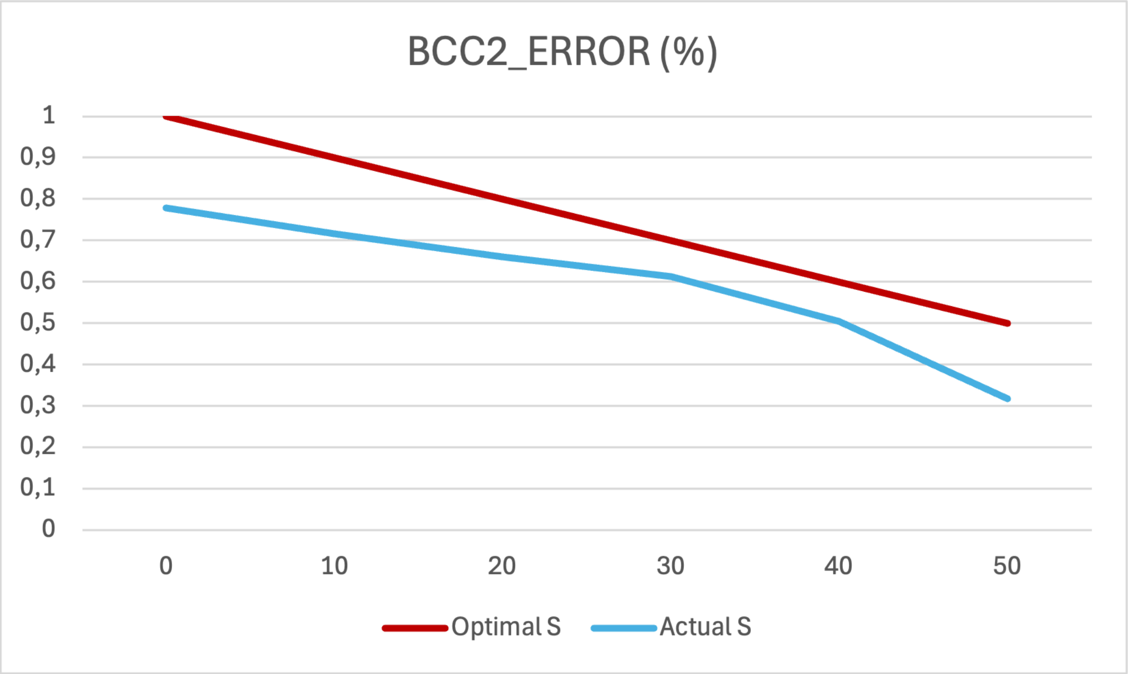
### Varying Propagation Delay



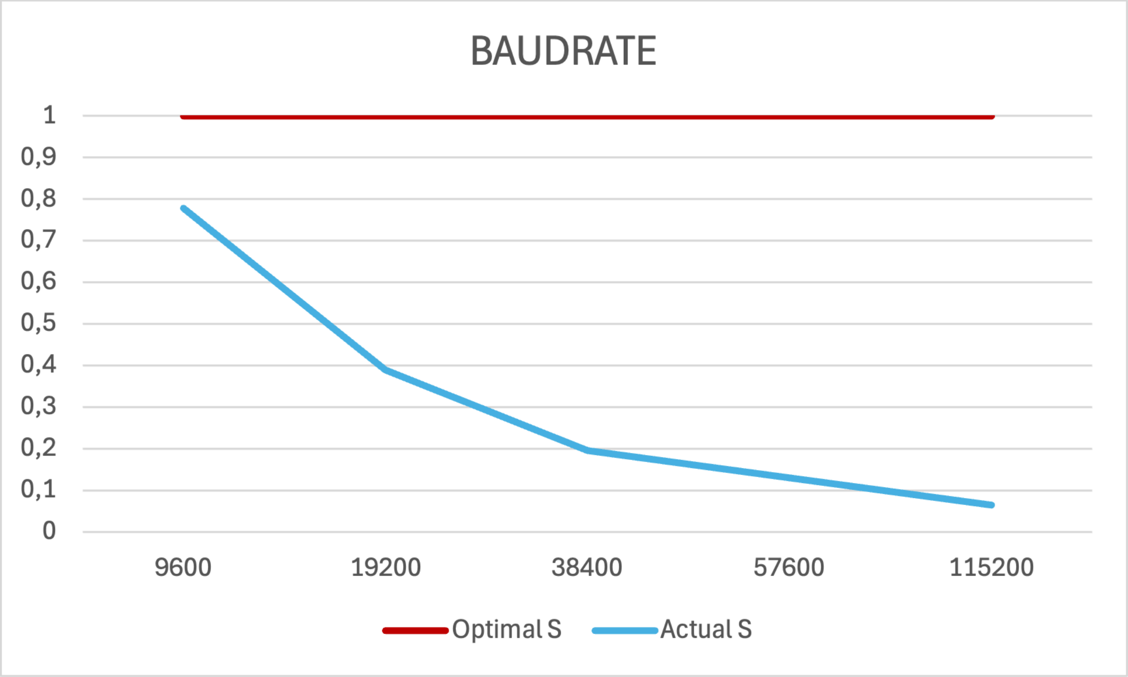
### Varying BCC1 Error Probability



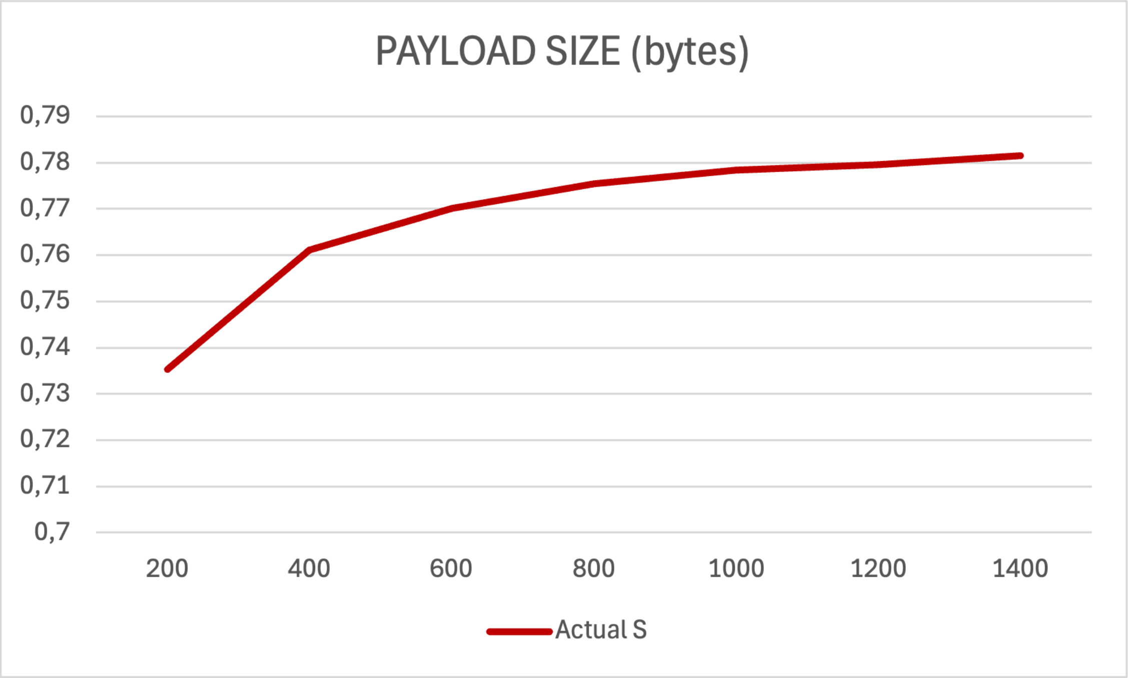
### Varying BCC2 Error Probability



### Varying Baud Rate



**Varying Payload size**



# Appendix 2 - Source Code

**serial_port.h**

(code provided, not modified)

```
// Serial port header.
// NOTE: This file must not be changed.

#ifndef _SERIAL_PORT_H_
#define _SERIAL_PORT_H_

// Open and configure the serial port.
// Returns -1 on error.
int openSerialPort(const char *serialPort, int baudRate);

// Restore original port settings and close the serial port.
// Returns -1 on error.
int closeSerialPort();

// Wait up to 0.1 second (VTIME) for a byte received from the serial
port (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);

// 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);

#endif // _SERIAL_PORT_H_
```

**serial_port.c**

(code provided, not modified)

```
// 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 fd = -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;
    fd = open(serialPort, oflags);
    if (fd < 0)
    {
        perror(serialPort);
        return -1;
    }

    // Save current port settings
    if (tcgetattr(fd, &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] = 0;  // Byte by byte
    // if read 0 bytes, is not an error, just didnt read anything,
should skip

    tcflush(fd, TCIOFLUSH);
    // Set new port settings
    if (tcsetattr(fd, TCSANOW, &newtio) == -1)
    {
        perror("tcsetattr");
        close(fd);
        return -1;
    }

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

    // Done
    return fd;
}

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

    return close(fd);
}

// Wait up to 0.1 second (VTIME) for a byte received from the serial
port (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(fd, 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(fd, bytes, numBytes);
}
```

**protocol.h**

```
// Data Link Protocol

#ifndef _PROTOCOL_H_
#define _PROTOCOL_H_

#define FLAG        0x7E
#define ESC         0x7D
#define A_T         0x03
#define A_R         0x01
#define C_SET       0x03
#define C_UA        0x07
#define C_DISC      0x0B
#define SUF_FLAG    0x5E
#define SUF_ESC     0x5D

#define C_INF(N)    ((N) ? 0x80 : 0x00)
#define C_RR(Nr)    (0xAA | Nr)
#define C_REJ(Nr)   (0x54 | Nr)

// Packet Control Field
#define C_START 1
#define C_DATA 2
#define C_END 3

// Packet Type Field
#define T_FILESIZE 0
#define T_FILENAME 1

#endif // _PROTOCOL_H_
```

**application_layer.h**

(code provided, not modified)

```
// Application layer protocol header.
// NOTE: This file must not be changed.

#ifndef _APPLICATION_LAYER_H_
#define _APPLICATION_LAYER_H_

// Application layer main function.
// Arguments:
//   serialPort: Serial port name (e.g., /dev/ttyS0).
//   role: Application role {"tx", "rx"}.
//   baudrate: Baudrate of the serial port.
//   nTries: Maximum number of frame retries.
//   timeout: Frame timeout.
//   filename: Name of the file to send / receive.
void applicationLayer(const char *serialPort, const char *role, int
baudRate,
                      int nTries, int timeout, const char *filename);

#endif // _APPLICATION_LAYER_H_
```

**application_layer.c**

```
// Application layer protocol implementation

#include "application_layer.h"
#include "link_layer.h"
#include "protocol.h"
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

#define MAX_FILENAME 100
#define METADATA_SIZE 20

int readPacketControl(unsigned char *buff, int *isEnd);
int readPacketData(unsigned char *buff, size_t *newSize, unsigned char
*dataPacket);
int sendPacketControl(unsigned char C, const char *filename, size_t
file_size);
int sendPacketData(size_t nBytes, unsigned char *data);
unsigned char * sizetouchar(size_t value, unsigned char *size);
size_t uchartosize (unsigned char n, unsigned char * numbers);
```

```
int sequenceNumber = 0;
size_t totalBytesRead = 0;


void applicationLayer(const char *serialPort, const char *role, int
baudRate,
                      int nTries, int timeout, const char *filename)
{
    if(serialPort == NULL || role == NULL || filename == NULL){
        printf("[ERROR] Initialization error: One or more required
arguments are NULL\n");
        return;
    }

    if (strlen(filename) > MAX_FILENAME) {
        printf("[ALERT] The lenght of the given file name is greater
than what is supported: %d characters'\n", MAX_FILENAME);
        return;
    }

    LinkLayer connectionParametersApp = {
        .baudRate = baudRate,
        .nRetransmissions = nTries,
        .timeout = timeout
    };

    strcpy(connectionParametersApp.serialPort, serialPort);
    connectionParametersApp.role = strcmp(role, "tx") == 0 ? LlTx :
LlRx;

    if (llopen(connectionParametersApp) == -1) {
        printf("[ERROR] Link layer error: Failed to open the
connection\n");
        return;
    }

    if (connectionParametersApp.role == LlTx) {
        size_t bytesRead = 0;
        unsigned char *buffer = (unsigned char *)
malloc(MAX_PAYLOAD_SIZE + METADATA_SIZE);
        if(buffer == NULL) {
            printf("[ERROR] Memory allocation error at buffer
creation\n");
```

```
            llclose(FALSE);
            return;
        }

        FILE* file = fopen(filename, "rb");
        if(file == NULL) {
            printf("[ERROR] File error: Unable to open the file for
reading\n");
            fclose(file);
            free(buffer);
            llclose(FALSE);
            return;
        }

        fseek(file, 0, SEEK_END);
        size_t file_size = ftell(file);
        rewind(file);

        printf("[INFO] Started sending file: '%s'\n", filename);
        if(sendPacketControl(C_START, filename, file_size) == -1) {
            printf("[ERROR] Transmission error: Failed to send the START
packet control\n");
            fclose(file);
            llclose(FALSE);
            return;
        }

        while ((bytesRead = fread(buffer, 1, MAX_PAYLOAD_SIZE, file)) >
0) {

            if(sendPacketData(bytesRead, buffer) == -1){
                printf("[ERROR] Transmission error: Failed to send the
DATA packet control\n");
                fclose(file);
                llclose(FALSE);
                return;
            }
        }

        if(sendPacketControl(C_END, filename, file_size) == -1){
            printf("[ERROR] Transmission error: Failed to send the END
packet control\n");
            fclose(file);
            llclose(FALSE);
            return;
        }
        printf("[INFO] Finished sending file: '%s'\n", filename);

        fclose(file);
    }

    if (connectionParametersApp.role == LlRx) {
        unsigned char * buf = malloc(MAX_PAYLOAD_SIZE + METADATA_SIZE);
        unsigned char * packet = malloc(MAX_PAYLOAD_SIZE +
METADATA_SIZE);

        if(buf == NULL || packet == NULL){
            printf("[ERROR] Initialization error: One or more buffers
pointers are NULL\n");
            llclose(FALSE);
            return;
        }

        FILE *file = fopen(filename, "wb");

        if(file == NULL) {
            printf("[ERROR] File error: Unable to open the file for
writing\n");
            fclose(file);
            llclose(FALSE);
            return;
        }

        size_t bytes_readed = 0;
        int isEnd = FALSE;

        while(!isEnd){

            if((bytes_readed = llread(buf)) == -1) {
                printf("[ERROR] Link layer error: Failed to read from
the link\n");
                fclose(file);
                llclose(FALSE);
                return;
            }

            if(buf[0] == C_START || buf[0] == C_END){
                if(readPacketControl(buf, &isEnd) == -1) {
                    printf("[ERROR] Packet error: Failed to read control
packet\n");
                    fclose(file);
                    llclose(FALSE);
                    return;
                }

            } else if(buf[0] == C_DATA){

                if(readPacketData(buf, &bytes_readed, packet) == -1) {
                    printf("[ERROR] Packet error: Failed to read data
packet\n");
                    fclose(file);
                    llclose(FALSE);
                    return;
                }
                fwrite(packet, 1, bytes_readed, file);
                totalBytesRead += bytes_readed;
            }
        }

        fclose(file);
    }


    if (llclose(TRUE) == -1) {
        printf("[ERROR] Link layer error: Failed to close the
connection\n");
        return;
    }

    printf("[SUCCESS] Connection closed successfully\n");
}


////////////////////////////////////////////////
// AUXILIARY FUNCTIONS
////////////////////////////////////////////////

int readPacketControl(unsigned char *buff, int *isEnd)
{
    if (buff == NULL) return -1;
    size_t pos = 0;

    if(buff[pos] == C_END) *isEnd = TRUE;
    else if (buff[pos] != C_START) return -1;

    // file size (V1)
    pos++;
    if (buff[pos++] != T_FILESIZE) return -1;
    unsigned char L1 = buff[pos++]; // V1 field size

    unsigned char * V1 = malloc(L1);
    if(V1 == NULL) return -1;

    memcpy(V1, buff + pos, L1);
    pos += L1;

    size_t file_size = uchartosize(L1, V1);
    free(V1);

    // name (V2)
    if(buff[pos++] != T_FILENAME) return -1;
    unsigned char L2 = buff[pos++]; // V2 field size

    char * file_name = malloc(MAX_FILENAME);
    if(file_name == NULL) return -1;

    memcpy(file_name, buff + pos, L2);
    file_name[L2] = '\0';


    if(buff[0] == C_START){
        printf("[INFO] Started receiving file: '%s'\n", file_name);
    } else if(buff[0] == C_END){
        if (file_size != totalBytesRead) {
            printf("[Warning] The received file size doesn't match the original file\n");
        }

        printf("[INFO] Finished receiving file: '%s'\n", file_name);
    }

    free(file_name);
    return 1;
}
```

```
int readPacketData(unsigned char *buff, size_t *newSize, unsigned char
*dataPacket)
{
    if (buff == NULL) return -1;
    if (buff[0] != C_DATA) return -1;

    *newSize = buff[2] * 256 + buff[3];
    memcpy(dataPacket, buff + 4, *newSize);

    return 1;
}

int sendPacketControl(unsigned char C, const char *filename, size_t
file_size)
{
    if(filename == NULL) return -1;

    unsigned char L1 = 0;
    unsigned char * V1 = sizetouchar(file_size, &L1);
    if(V1 == NULL) return -1;

    unsigned char L2 = (unsigned char) strlen(filename);

    unsigned char *packet = (unsigned char *) malloc(5 + L1 + L2);
    if(packet == NULL) {
        free(V1);
        return -1;
    }

    size_t pos = 0;
    packet[pos++] = C;

    // file size (V1)
    packet[pos++] = T_FILESIZE;
    packet[pos++] = L1;
    memcpy(packet + pos, V1, L1);
    pos += L1;
    free(V1);

    // file name (V2)
    packet[pos++] = T_FILENAME;
    packet[pos++] = L2;
    memcpy(packet + pos, filename, L2);
    pos += L2;

    int result = llwrite(packet, (int) pos);

    free(packet);
    return result;
}

int sendPacketData(size_t nBytes, unsigned char *data)
{
    if(data == NULL) return -1;

    unsigned char *packet = (unsigned char *) malloc(nBytes + 4);
    if(packet == NULL) return -1;

    packet[0] = C_DATA;
    packet[1] = (sequenceNumber++) % 100;
    packet[2] = nBytes >> 8;
    packet[3] = nBytes & 0xFF;

    memcpy(packet + 4, data, nBytes);

    int result = llwrite(packet, nBytes + 4);

    free(packet);
    return result;
}

// Function to convert a size_t value to an array of unsigned char
(octets)
/**
 * @brief Converts a size_t value to an array of unsigned char (octets).
 *
 * This function converts a size_t value to an array of unsigned char
(octets) and
 * returns the array. The length of the array is stored in the variable
pointed to by size.
 *
 * @param value The size_t value to be converted.
 * @param size Pointer to an unsigned char where the length of the array
will be stored.
 * @return unsigned char* Pointer to the array of octets, or NULL if
memory allocation fails.
 */
```

```c
unsigned char * sizetouchar(size_t value, unsigned char *size)
{
    if (size == NULL) return NULL;

    size_t temp = value, l = 0;

    do {
        l++;
        temp >>= 8;
    } while (temp);

    unsigned char *bytes = malloc(l);
    if (bytes == NULL) return NULL;

    for (size_t i = 0; i < l; i++) {
        bytes[i] = value & 0xFF;
        value >>= 8;
    }

    *size = l;
    return bytes;
}

// Function to convert an array of unsigned char (octets) to a size_t value
/**
 * @brief Converts an array of unsigned char (octets) to a size_t value.
 *
 * This function converts an array of unsigned char (octets) to a size_t value.
 *
 * @param n The number of octets in the array.
 * @param numbers Pointer to the array of unsigned char (octets).
 * @return size_t The converted size_t value.
 */
size_t uchartosize (unsigned char n, unsigned char * numbers)
{
    if(numbers == NULL) return 0;

    size_t value = 0;
    size_t power = 1;

    for(int i = 0; i < n; i++) {
        value += numbers[i] * power;
        power <<= 8;
    }

    return value;
}
```

**link_layer.h**

(code provided, not modified)

```
// Link layer header.
// NOTE: This file must not be changed.

#ifndef _LINK_LAYER_H_
#define _LINK_LAYER_H_

typedef enum
{
    LlTx,
    LlRx,
} LinkLayerRole;

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

// SIZE of maximum acceptable payload.
// Maximum number of bytes that application layer should send to link
layer
#define MAX_PAYLOAD_SIZE 1000

// MISC
#define FALSE 0
#define TRUE 1

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

```c
// 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);

#endif // _LINK_LAYER_H_
```

**link_layer.c**

```c
// Link layer protocol implementation

#include "link_layer.h"
#include "serial_port.h"
#include "protocol.h"
#include "statistics.h"

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

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

typedef enum {
    START_STATE,
    FLAG_RCV,
    A_RCV,
    C_RCV,
    BCC_OK,
    STOP_STATE,
    DATA_STATE,
    ESC_STATE
} LinkLayerState;

void alarmHandler(int signal);
void alarmDisable();
void nextNs();
void nextNr();
void showStatisticsTerminal();
int destuffing(unsigned char *buf, int bufSize, int *newSize, unsigned
char *BCC2);
int sendCommandFrame(unsigned char A, unsigned char C);
int receiveFrame(unsigned char A_EXPECTED, unsigned char C_EXPECTED);
int receiveRetransmissionFrame(unsigned char A_EXPECTED, unsigned char
C_EXPECTED, unsigned char A_SEND, unsigned char C_SEND);

int alarmEnabled = FALSE;
int alarmCount = 0;
int RETRANSMISSIONS = 0;
int TIMEOUT = 0;
LinkLayerRole ROLE;
int BAUDRATE;
unsigned char C_Ns = 0;
unsigned char C_Nr = 0;

Statistics statistics = {0, 0, 0, 0.0};

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

    BAUDRATE = connectionParameters.baudRate;
    ROLE = connectionParameters.role;
    RETRANSMISSIONS = connectionParameters.nRetransmissions;
    TIMEOUT = connectionParameters.timeout;

    switch (ROLE) {
        case LlTx:

            if (receiveRetransmissionFrame(A_T, C_UA, A_T, C_SET) != 1)
return -1;
            gettimeofday(&statistics.startTime, NULL);
            statistics.nFrames++;

            printf("[STATUS] Connection Established!\n");

            break;

        case LlRx:

            if (receiveFrame(A_T, C_SET) != 1) return -1;
            gettimeofday(&statistics.startTime, NULL);
            statistics.nFrames++;
            statistics.bytesRead += 5;
            srand(time(NULL)); // seed random number generator

            if (sendCommandFrame(A_T, C_UA) != 1) return -1;

            printf("[STATUS] Connection Established!\n");

            break;
    }

    return 1;
}

////////////////////////////////////////////////
// LLWRITE
////////////////////////////////////////////////
int llwrite(const unsigned char *buf, int bufSize)
{
    if (buf == NULL) return -1;

    int frameSize = bufSize + 6;
    unsigned char *frame = malloc(frameSize);

    // Create frame header
    frame[0] = FLAG;
    frame[1] = A_T;
    frame[2] = C_Ns ? C_INF(1) : C_INF(0);
    frame[3] = frame[1] ^ frame[2];
    memcpy(frame + 4, buf, bufSize);

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

    // Data and stuffing
    int pos = 4;
    for (int i = 0; i < bufSize; i++) {
        switch (buf[i]) {
            case FLAG:
                frame = realloc(frame, ++frameSize);
                frame[pos++] = ESC;
                frame[pos++] = SUF_FLAG;
                break;

            case ESC:
                frame = realloc(frame, ++frameSize);
                frame[pos++] = ESC;
                frame[pos++] = SUF_ESC;
                break;

            default:
                frame[pos++] = buf[i];
                break;
        }
    }

    // tail
    frame[pos++] = BCC2;
    frame[pos++] = FLAG;


    // Send frame
    LinkLayerState state = START_STATE;

    (void)signal(SIGALRM, alarmHandler);

    if (writeBytesSerialPort(frame, frameSize) < 0) {
        free(frame);
        printf("[ERROR] Error writing send command\n");
        return -1;
    }
    alarm(TIMEOUT);

    unsigned char byte_C = 0, byte_A = 0;

    while (state != STOP_STATE && alarmCount <= RETRANSMISSIONS)
    {
        int result;
        unsigned char byte;

        if ((result = readByteSerialPort(&byte)) < 0) {
            free(frame);
            printf("[ERROR] Error reading response\n");
            return -1;
        }

        else if (result > 0) {
            switch (state) {

                case START_STATE:
                    byte_C = 0;
                    byte_A = 0;
                    if (byte == FLAG) state = FLAG_RCV;
                    break;

                case FLAG_RCV:
                    if (byte == A_R || byte == A_T) {
                        state = A_RCV;
                        byte_A = byte;
                    }
                    else if (byte != FLAG) state = START_STATE;
                    break;

                case A_RCV:
                    if (byte == C_RR(0) || byte == C_RR(1) || byte ==
C_REJ(0) || byte == C_REJ(1)) {
                        state = C_RCV;
                        byte_C = byte;
                    }
                    else if (byte == FLAG) state = FLAG_RCV;
                    else state = START_STATE;
                    break;

                case C_RCV:
```

```
                if (byte == FLAG) state = FLAG_RCV;
                else if ((byte_A ^ byte_C) == byte) state = BCC_OK;
                else state = START_STATE;
                break;

            case BCC_OK:
                if (byte == FLAG) state = STOP_STATE;
                else state = START_STATE;
                break;

            default:
                state = START_STATE;

        }
    }

    if (state == STOP_STATE) {

        if (byte_C == C_REJ(0) || byte_C == C_REJ(1)) {
            alarmEnabled = TRUE;
            alarmCount = 0;
            printf("[ALERT] Frame rejected, resending frame\n");
        }

        else if (byte_C == C_RR(0) || byte_C == C_RR(1)) {
            statistics.nFrames++;

            alarmDisable();
            nextNs();
            free(frame);
            return bufSize;
        }

    }

    if (alarmEnabled) {

        alarmEnabled = FALSE;

        if (alarmCount <= RETRANSMISSIONS) {
            if (writeBytesSerialPort(frame, frameSize) < 0) {
                printf("[ERROR] Error writing send command\n");
                return -1;
            }
```

```
                alarm(TIMEOUT);
            }

            state = START_STATE;
        }
    }

    alarmDisable();
    free(frame);

    return -1;
}

////////////////////////////////////////////////
// LLREAD
////////////////////////////////////////////////
int llread(unsigned char *packet)
{
    usleep(TPROPAGATION * 1000); // simulate propagation delay in ms

    unsigned char byte_C = 0;
    int pos = 0;

    LinkLayerState state = START_STATE;

    while (state != STOP_STATE)
    {
        int result;
        unsigned char byte = 0;

        if ((result = readByteSerialPort(&byte)) < 0) {
            printf("[ERROR] Error reading response\n");
            return -1;
        }

        else if (result > 0) {
            switch (state) {

                case START_STATE:
                    pos = 0;
                    byte_C = 0;
                    if (byte == FLAG) state = FLAG_RCV;
                    break;
                case FLAG_RCV:
                    if (byte == A_T) state = A_RCV;
                    else if (byte != FLAG) state = START_STATE;
                    break;

                case A_RCV:
                    if (byte == C_INF(0) || byte == C_INF(1)) {
                        state = C_RCV;
                        byte_C = byte;
                    }
                    else if (byte == FLAG) state = FLAG_RCV;
                    else state = START_STATE;
                    break;

                case C_RCV:
                    if ((A_T ^ byte_C) == byte) state = DATA_STATE;
                    else {
                        if (byte == FLAG) state = FLAG_RCV;
                        else state = START_STATE;
                        statistics.errorFrames++;
                    }
                    break;

                case DATA_STATE:
                    if (byte == FLAG) {
                        int newSize = 0;
                        unsigned char BCC2 = 0;

                        if (destuffing(packet, pos, &newSize, &BCC2) !=
1) {

                            printf("[ERROR] Error destuffing\n");
                            return -1;
                        }

                        // create BCC2
                        unsigned char xor = packet[0];
                        for (int i = 1; i < newSize; i++) {
                            xor ^= packet[i];
                        }

                        unsigned char C_;   // the response frame

                        if (xor == BCC2) {
                            // BCC2 correct, send a positive
acknowledgment (RR)
                            C_ = (byte_C == C_INF(0)) ? C_RR(1) :
C_RR(0);
                        }

                        else {
                            // BCC2 incorrect
                            if ((C_Nr == 0 && byte_C == C_INF(1)) ||
(C_Nr == 1 && byte_C == C_INF(0))) {
                                // received frame is the expected, send
a positive acknowledgment (RR)
                                C_ = (byte_C == C_INF(0)) ? C_RR(1) :
C_RR(0);
                            } else {
                                // received frame is not the expected,
send a negative acknowledgment (REJ)
                                C_ = (byte_C == C_INF(0)) ? C_REJ(0) :
C_REJ(1);
                            }
                        }

                        // Simulate probability of error in BCC1 and
BCC2
                        // Use only for testing purposes
                        if ((C_Nr == 0 && byte_C == C_INF(0)) || (C_Nr
== 1 && byte_C == C_INF(1))) {
                            if (rand() % 100 <= BCC1_ERROR - 1) {
                                statistics.errorFrames++;
                                break;
                            }

                            if (rand() % 100 <= BCC2_ERROR - 1) C_ =
(byte_C == C_INF(0)) ? C_REJ(0) : C_REJ(1);

                        }

                        usleep(TPROPAGATION * 1000); // simulate
propagation delay in ms

                        if (sendCommandFrame(A_R, C_) != 1) {
                            printf("[ERROR] Error sending response\n");
                            return -1;
                        }
                        state = START_STATE;
                        if (C_ == C_REJ(0) || C_ == C_REJ(1)) {
                            statistics.errorFrames++;
                            printf("[ALERT] Frame rejected, resending
frame\n");
                            break;
                        }

                        // update sequence number
                        if ((C_Nr == 0 && byte_C == C_INF(0)) || (C_Nr
== 1 && byte_C == C_INF(1))) {
                            statistics.bytesRead += newSize + 6;
                            statistics.nFrames++;

                            nextNr();
                            return newSize;
                        }

                        //printf("[ERROR] Discarding frame,
duplicate\n");
                    } else {
                        packet[pos++] = byte;
                    }

                    break;

                default:
                    state = START_STATE;

            }
        }
    }

    return -1;
}

////////////////////////////////////////////////
// LLCLOSE
////////////////////////////////////////////////
int llclose(int showStatistics)
{
    switch (ROLE) {
        case LlTx:
            if (receiveRetransmissionFrame(A_R, C_DISC, A_T, C_DISC) !=
1) return -1;
            statistics.nFrames++;

            if (sendCommandFrame(A_R, C_UA) != 1) return
closeSerialPort();
            statistics.nFrames++;

            break;

        case LlRx:
            if (receiveFrame(A_T, C_DISC) == 1) {
                statistics.nFrames++;
                statistics.bytesRead += 5;

                if (sendCommandFrame(A_R, C_DISC) != 1) return -1;
                statistics.nFrames++;
                statistics.bytesRead += 5;
            }

            break;

        default:
            return -1;

    }

    gettimeofday(&statistics.endTime, NULL);

    if (showStatistics) {
        showStatisticsTerminal();
    }

    return closeSerialPort();
}


////////////////////////////////////////////////
// AUXILIARY FUNCTIONS
////////////////////////////////////////////////

// Alarm handler
void alarmHandler(int signal)
```

```
{
    printf("Alarm #%d\n", alarmCount + 1);
    alarmCount++;
    alarmEnabled = TRUE;
    statistics.retransmissions++;
}

// Disable alarm
void alarmDisable()
{
    alarm(0);
    alarmEnabled = FALSE;
    alarmCount = 0;
}

// Switch Ns between 0 and 1
void nextNs()
{
    C_Ns = C_Ns ? 0 : 1;
}

// Switch Nr between 0 and 1
void nextNr()
{
    C_Nr = C_Nr ? 0 : 1;
}

/**
 * @brief Perform byte destuffing on the input buffer.
 *
 * This function processes the input buffer to remove escape sequences
and
 * reconstruct the original data. It also extracts the BCC2 (Block Check
Character 2)
 * from the buffer.
 *
 * @param buf The input buffer containing the stuffed data.
 * @param bufSize The size of the input buffer.
 * @param newSize Pointer to an integer where the new size of the buffer
will be stored.
 * @param BCC2 Pointer to an unsigned char where the BCC2 will be
stored.
 * @return int Returns 1 on success, -1 on error (e.g., null pointers),
1 on invalid buffer size.
```

```
 */
int destuffing(unsigned char *buf, int bufSize, int *newSize, unsigned
char *BCC2)
{
    if (buf == NULL || newSize == NULL) return -1;
    if (bufSize < 1) return 1;

    unsigned char *r = buf, *w = buf;

    while (r < buf + bufSize) {
        if (*r != ESC) *w++ = *r++; // if not escape, copy byte
        else {
            // if ESC, check next and replace with FLAG/ESC
            if (*(r + 1) == SUF_FLAG) *w++ = FLAG;
            else if (*(r + 1) == SUF_ESC) *w++ = ESC;
            r += 2;
        }
    }

    *BCC2 = *(w - 1);
    *newSize = w - buf - 1;

    return 1;
}

// Send Supervision Frame and Unnumbered Frame
// Returns 1 on success, -1 on error
int sendCommandFrame(unsigned char A, unsigned char C)
{
    unsigned char buf_T[5] = {FLAG, A, C, A ^ C, FLAG};

    return (writeBytesSerialPort(buf_T, 5) < 0) ? -1 : 1;
}

// Receive Frame and check if it is the expected frame
int receiveFrame(unsigned char A_EXPECTED, unsigned char C_EXPECTED)
{
    LinkLayerState state = START_STATE;

    while (state != STOP_STATE)
    {
        int result;
        unsigned char byte = 0;
        if((result = readByteSerialPort(&byte)) < 0) {
            printf("[ERROR] Error reading response\n");
            return -1;
        }

        else if(result > 0){
            switch (state) {

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

                case FLAG_RCV:
                    if (byte == A_EXPECTED) state = A_RCV;
                    else if (byte != FLAG) state = START_STATE;
                    break;

                case A_RCV:
                    if (byte == C_EXPECTED) state = C_RCV;
                    else if (byte == FLAG) state = FLAG_RCV;
                    else state = START_STATE;
                    break;

                case C_RCV:
                    if (byte == FLAG) state = FLAG_RCV;
                    else if ((A_EXPECTED ^ C_EXPECTED) == byte) state =
BCC_OK;

                    else state = START_STATE;
                    break;

                case BCC_OK:
                    if (byte == FLAG) state = STOP_STATE;
                    else state = START_STATE;
                    break;

                default:
                    state = START_STATE;

            }
        }
    }

    return 1;
}
```

```
// Receive Frame with retransmission and check if it is the expected
frame
// Returns 1 on success, -1 on error
int receiveRetransmissionFrame(unsigned char A_EXPECTED, unsigned char
C_EXPECTED, unsigned char A_SEND, unsigned char C_SEND)
{
    LinkLayerState state = START_STATE;

    (void)signal(SIGALRM, alarmHandler);

    if (sendCommandFrame(A_SEND, C_SEND) != 1) return -1;

    alarm(TIMEOUT);

    while (state != STOP_STATE && alarmCount <= RETRANSMISSIONS)
    {
        int result;
        unsigned char byte = 0;

        if((result = readByteSerialPort(&byte)) < 0) {
            printf("[ERROR] Error reading UA frame\n");
            return -1;
        }

        else if(result > 0){
            switch (state) {

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

                case FLAG_RCV:
                    if (byte == A_EXPECTED) state = A_RCV;
                    else if (byte != FLAG) state = START_STATE;
                    break;

                case A_RCV:
                    if (byte == C_EXPECTED) state = C_RCV;
                    else if (byte == FLAG) state = FLAG_RCV;
                    else state = START_STATE;
                    break;

                case C_RCV:
                    if (byte == FLAG) state = FLAG_RCV;
                    else if ((C_EXPECTED ^ A_EXPECTED) == byte) state =
BCC_OK;
                    else state = START_STATE;
                    break;

                case BCC_OK:
                    if (byte == FLAG) state = STOP_STATE;
                    else state = START_STATE;
                    break;

                default:
                    state = START_STATE;

            }
        }

        if (state == STOP_STATE) {
            alarmDisable();
            return 1;
        }

        else if (alarmEnabled) {
            alarmEnabled = FALSE;

            if (alarmCount <= RETRANSMISSIONS) {

                if (sendCommandFrame(A_SEND, C_SEND) != 1) {
                    printf("[ERROR] Error writing send command\n");
                    return -1;
                }

                alarm(TIMEOUT);
            }

            state = START_STATE;
        }
    }

    alarmDisable();

    return -1;
}
```

```
void showStatisticsTerminal() {
    const char *role_str = (ROLE == LlTx) ? "TRANSMITTER" : "RECEIVER";
    printf("\n\t======= [%s STATISTICS] =======\n\n", role_str);
    if (ROLE == LlTx) { // Transmitter
        printf("               Good frames sent: %u frames\n",
statistics.nFrames);
        printf("          Total retransmissions: %u\n",
statistics.retransmissions);
        printf("              Image Upload time: %f seconds\n",
timeDiff(statistics.startTime, statistics.endTime));
        printf("\n");
        printf("              Actual efficiency: %f\n",
actual_efficiency(statistics, BAUDRATE));
        printf("             Optimal efficiency: %f\n",
optimal_efficiency(BAUDRATE, MAX_PAYLOAD_SIZE));
    } else {        // Receiver
        printf("           Good frames received: %u frames\n",
statistics.nFrames);
        printf("           Bad frames discarded: %u frames\n",
statistics.errorFrames);
        printf("     Received bytes (destuffed): %u bytes\n",
statistics.bytesRead);
        printf("            Image Download time: %f seconds\n",
timeDiff(statistics.startTime, statistics.endTime));
        printf("\n");
        printf("              Received bit rate: %f bits/s\n",
received_bit_rate(statistics));
    }
    printf("\n\t=====================================");
    if (ROLE == LlTx) printf("===");
    printf("\n\n");
}
```

### statistics.h

```
#ifndef _STATISTICS_H_
#define _STATISTICS_H_

#include <stdio.h>
#include <sys/time.h>

#define TPROPAGATION    0  // propagation delay in ms
#define BCC1_ERROR      0   // percentage % of frames with BCC1 error
#define BCC2_ERROR      0   // percentage % of frames with BCC2 error
```

```
#define FILESIZE        10968

typedef struct {
    unsigned int bytesRead;
    unsigned int nFrames;
    unsigned int errorFrames;
    unsigned int retransmissions;
    struct timeval startTime;
    struct timeval endTime;
} Statistics;

double timeDiff(struct timeval start, struct timeval end);

double propagation_to_transmission_ratio(int baudrate, int maxPayload);

double received_bit_rate(Statistics stats);

double fer();

double optimal_efficiency(int baudrate, int maxPayload);

double actual_efficiency(Statistics stats, int baudrate);

#endif // _STATISTICS_H_
```

**statistics.c**

```
#include "statistics.h"

// Calculate the difference between two timeval structs in seconds.
double timeDiff(struct timeval start, struct timeval end) {
    return (end.tv_sec - start.tv_sec) + (end.tv_usec - start.tv_usec) /
1000000.0;
}

// a
double propagation_to_transmission_ratio(int baudrate, int maxPayload) {
    return ((double) TPROPAGATION / 1000.0) / ((double) maxPayload * 8.0
/ (double) baudrate);
}

double received_bit_rate(Statistics stats) {
    return (double) (FILESIZE * 8) / timeDiff(stats.startTime,
stats.endTime);
```

```c
}

// FER = P(bcc1 error) + P(bcc2 error) * (1 - P(bcc1 error))
// probability of any error in either BCC1 or BCC2
double fer() {
    double bcc1_error_rate = (double) BCC1_ERROR / 100.0;
    double bcc2_error_rate = (double) BCC2_ERROR / 100.0;
    return bcc1_error_rate + bcc2_error_rate * (1 - bcc1_error_rate);
}

// Optimal Efficiency = (1 - FER) / (1 + 2a)
double optimal_efficiency(int baudrate, int maxPayload) {
    double fer_value = fer();
    double a = propagation_to_transmission_ratio(baudrate, maxPayload);
    return (1 - fer_value) / (1 + 2 * a);
}

// Actual Efficiency =  Actual Received Bitrate / Link Capacity
double actual_efficiency(Statistics stats, int baudrate) {
    return (double) (received_bit_rate(stats) / baudrate);
}
```

**main.c**

(code provided, not modified)

```c
// Main file of the serial port project.
// NOTE: This file must not be changed.

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "application_layer.h"

#define N_TRIES 3
#define TIMEOUT 4


// Arguments:
//   $1: /dev/ttySxx
//   $2: baud rate
//   $3: tx | rx
//   $4: filename
int main(int argc, char *argv[])
```

```
{
    if (argc < 5) {
        printf("Usage: %s /dev/ttySxx baudrate tx|rx filename\n",
argv[0]);
        exit(1);
    }

    const char *serialPort = argv[1];
    const int baudrate = atoi(argv[2]);
    const char *role = argv[3];
    const char *filename = argv[4];

    // Validate baud rate
    switch (baudrate) {
        case 1200:
        case 1800:
        case 2400:
        case 4800:
        case 9600:
        case 19200:
        case 38400:
        case 57600:
        case 115200:
            break;
        default:
            printf("Unsupported baud rate (must be one of 1200, 1800,
2400, 4800, 9600, 19200, 38400, 57600, 115200)\n");
            exit(2);
    }

    // Validate role
    if (strcmp("tx", role) != 0 && strcmp("rx", role) != 0) {
        printf("ERROR: Role must be \"tx\" or \"rx\"\n");
        exit(3);
    }

    printf("Starting link-layer protocol application\n"
           "  - Serial port: %s\n"
           "  - Role: %s\n"
           "  - Baudrate: %d\n"
           "  - Number of tries: %d\n"
           "  - Timeout: %d\n"
           "  - Filename: %s\n",
           serialPort,
           role,
           baudrate,
           N_TRIES,
           TIMEOUT,
           filename);

    applicationLayer(serialPort, role, baudrate, N_TRIES, TIMEOUT,
filename);

    return 0;
}
```