RCOM

LEIC - 2024/2025

1st Laboratory Work

Data Link Protocol Implementation

Authors:

Bruno Ferreira up202207863@fe.up.pt

Pedro Marinho up202206854@fe.up.pt

Summary

This project focuses on developing a Stop-and-Wait Automatic Repeat reQuest (ARQ) protocol implemented as a basic data link layer for reliable point-to-point communication. The protocol is designed to operate under varying network conditions, including different baud rates, error rates (FER), frame sizes, and propagation delays. This report details the protocol's architecture, functionality, and validation.

Our findings show that the protocol maintains reliable communication even under adverse conditions, although efficiency is impacted by high FER and lower baud rates. The modular approach and retransmission mechanisms ensure data integrity while highlighting the trade-offs inherent in a Stop-and-Wait ARQ system.

Introduction

The objective of this work is to implement a data link protocol that ensures reliable transmission across a communication link. This protocol manages connection setup, data encapsulation, error detection, and disconnection. It uses the Stop-and-Wait ARQ mechanism to address potential data loss and corruption, making it suitable for simple yet robust communication.

Architecture

The protocol's architecture follows a layered design that isolates control and data handling logic, promoting modularity and clear interaction interfaces between functional components:

- **Data Link Layer (DLL):** Responsible for physical link management, ensuring reliable data transfer through functions like *llopen*, *llclose*, *llwrite*, and *llread*. The DLL uses a state machine to process each received byte, verifying the correctness of frames and determining their type, which supports structured handling of incoming data.
- **Application Layer:** Interfaces directly with the user, coordinating the setup and execution of data transmissions. It manages the creation and processing of data and control packets to facilitate smooth communication with the DLL.
- **Alarm System:** Implements a basic alarm mechanism in *alarm.c* to handle retransmission, enabling the system to resend data when expected acknowledgments are delayed. Upon a triggered signal, the alarm is disabled, and a counter is incremented to track alarm events.
- **State Machine:** Exclusively utilized within the DLL, this finite state machine processes bytes as they are read, determining if the frame is correctly received and classifying the frame type, ensuring accurate and ordered link-layer operations.

Functional Blocks and Interfaces

- **link_layer.c:** Implements core link-layer functions, handling low-level communication and data transfer functions essential for reliable data transmission over the physical link.

- **application_layer.c:** Oversees higher-level data transfer operations, preparing packets for transmission or handling received data to ensure consistency between user applications and the DLL.
- **state_machine.c:** Manages link-layer state transitions within the DLL. Each byte passed through the state machine facilitates orderly protocol states, maintaining consistent communication flow.
- **alarm.c:** Provides alarm functionality to support retransmission by setting an alarm signal, tracking its enablement status, and incrementing the retry count upon each trigger.

These functional blocks work in tandem, allowing each layer to fulfill its role independently yet in coordination, ensuring robust data transfer even under less-than-ideal conditions.

Code Structure

The implementation is organized into several key files, each dedicated to a specific aspect of the protocol's functionality:

- **link_layer.c**: Contains the implementation of essential data link functions that manage link setup, data transmission, acknowledgment handling, and link teardown.
    - **llopen**: Initializes the communication link by opening the specified serial port and configuring the connection parameters. It distinguishes between receiver and transmitter roles to establish the connection appropriately.
    - **llclose**: Closes the connection gracefully, ensuring no data is left untransmitted and that resources are released. It also allows for the option to display statistics about the connection.
    - **llwrite**: Segments data into I-frames, manages frame transmission, and handles acknowledgment. It implements a retry mechanism to ensure reliability by retransmitting frames until an acknowledgment is received or the maximum number of attempts is reached.
    - **llread**: Reads incoming data frames from the serial port, verifies their integrity using a state machine, and manages acknowledgment and retransmission logic. It can handle duplicate frames and frames with bad data, responding appropriately with acknowledgment (ACK) or rejection (REJ).
    - **Helper functions:** The following functions are helper functions that support the operations of the data link layer. They enhance code readability and maintainability: **send_SET, send_ACK, llopen_receiver, llopen_transmitter, safe_write, send_data_frame, send_RR, send_REJ, send_DISC, llclose_transmitter, llclose_receiver.**
- **application_layer.c**: Manages the creation and processing of packets, organizing control packets (e.g., START, END) and data packets to enable reliable data transfer. The ***send_control_packet*** function builds and sends control frames that signal the beginning and end of each transfer session, while the ***send_data_packets*** function organizes data into sequenced frames to facilitate orderly communication with the data link layer. Functions like ***process_packet*** and ***initialize_packet_buffer*** ensure correct handling and sequencing of data packets for seamless file transfer.

- **state_machine.c**: Implements a state machine used in the data link layer to process incoming bytes and determine whether a valid frame is being received. The state machine transitions between states based on specific events, such as the reception of valid bytes. The states include ***START***, ***FLAG_RCV***, ***A_RCV***, ***C_RCV***, and ***BCC1_OK***, which allow it to dynamically adapt to the received data. The state machine also manages data processing, acknowledgment handling, and error detection, ensuring robust communication in the data link layer.

- **alarm.c**: Provides timeout management by enabling retransmission for unacknowledged frames. The alarm mechanism sets a timer to trigger the alarm handler if no acknowledgment is received within a specified interval. When triggered, the handler re-enables the alarm for further retries and tracks the number of attempts to avoid infinite loops.

Key Data Structures

- **LinkLayer**: Stores configuration details for the link, including the serial port, link layer role (transmitter or receiver), baud rate, number of retransmissions, and timeout settings. This structure centralizes key parameters that are accessible across link-layer functions, facilitating easier management of communication settings.
- **Packet**: Represents a data packet, encapsulating key fields such as the sequence number, data payload, and data size. This structure is crucial for reliable data encapsulation and transfer, enabling effective session control and maintaining the integrity of the transmitted information.
- **state_machine**: This structure tracks the current state of the link layer during frame reception. It ensures proper synchronization between the sender and receiver by managing transitions through various states based on received bytes.

- **ll_statistics**: Tracks statistical information related to link layer operations, such as the number of sent and received frames, retransmissions, timeouts, and invalid frame counts. This structure provides valuable insights into the protocol's performance and reliability, allowing for better diagnostics and optimizations.

These files and structures collectively form a robust codebase, enabling each component to operate independently while contributing to the protocol's overall functionality, ensuring smooth data transfer even under challenging conditions.

Main Use Cases

The project facilitates communication between two computers through a serial port, utilizing a structured approach for data transmission. Users will compile the application using the provided Makefile and execute the program with the following command format: “./bin/main [port] [rx/tx] [file]”.

- **[port]**: The serial port number to establish the connection, represented as “/dev/ttySx”.
- **[rx/tx]**: A flag indicating whether the application operates in receiver (rx) or transmitter (tx) mode.

- **[file]**: The name of the file to be sent or received.

For example, to receive a file: “./bin/main /dev/ttyS0 rx penguin-received.gif”

And to transmit a file: “./bin/main /dev/ttyS0 tx penguin.gif”

The receiver should be initiated first, waiting for the transmitter to establish a connection. If the transmitter starts first, it will attempt to connect and, upon exceeding the maximum number of attempts, the application will terminate. Once the connection is established, the transmitter sends the file data while the receiver captures it, saving it with the specified filename. Progress updates, including transfer percentages and any error messages, will be displayed in the console.

Additionally, the project includes a virtual cable program to simulate the serial connection, allowing users to test the protocol under different conditions, such as with or without cable disconnections and noise. This enables users to ensure the integrity of the transmitted files, even in adverse conditions. The successful file transfer can be verified by comparing the original and received files using the “diff” command or the provided Makefile target.

Logical Link Protocol

The primary objectives of this link layer protocol include configuring the serial port, establishing a connection, data transmission (including byte stuffing and error handling), and error recovery. Key functions are as follows:

llopen and llclose

The *llopen* function initializes the serial port and begins the connection setup process. It distinguishes between the roles of receiver and transmitter. For transmitters, it sends a **SET** frame and waits for a **UA** acknowledgment, retransmitting the **SET** frame if no acknowledgment is received within a specified timeout, retrying up to the defined maximum number of times. If the maximum retransmissions are reached without success, the connection attempt is aborted, returning an error state. For receivers, *llopen_receiver* waits for the **SET** frame from the transmitter, processes the incoming bytes through a state machine, and responds with an **ACK** frame upon successful reception.

The *llclose* function gracefully terminates the connection by sending a **DISC** frame and awaiting the corresponding acknowledgment. Upon successful termination, *llclose* resets the serial port configuration to its initial state. The disconnection process involves sending a **DISC** command and handling potential retransmissions until the **DISC** is received successfully, which is tracked by the number of retransmissions and timeouts.

llwrite and llread

The *llwrite* function is responsible for preparing and transmitting data frames. It constructs the frame with appropriate flags, addresses, and data, applying byte stuffing to handle special byte values (like **FLAG** and **ESC**). It utilizes a state machine to manage the transmission state and handles retries based on acknowledgment feedback. If no

acknowledgment (**RR**) or a **REJ** frame is received, indicating an error, *llwrite* retransmits the frame.

On the receiver's end, *llread* manages incoming frames by performing byte destuffing, verifying frame integrity using **BCC2**, and responding with an acknowledgment (**RR**) or a rejection (**REJ**) based on the frame's integrity. Importantly, all frames read are processed by a state machine that determines the validity of each frame and dictates how the link layer should proceed. This state machine ensures that the receiver can identify valid frames and decide whether to accept, reject, or request retransmission of the data, thereby maintaining the integrity of the communication.

Data Handling

The protocol implements safe writing to the serial port using the *safe_write* function, which ensures that all requested bytes are written to the serial port, handling partial writes. This function continuously attempts to write remaining bytes until the total number of bytes written matches the request.

Connection Establishment/Termination

The connection establishment and termination processes are essential for reliable communication.

To start, the *send_SET* function sends a **SET** command to initiate the connection. Upon receiving the **SET** frame, the receiver responds with an acknowledgment using *send_ACK*, indicating readiness. The sender then waits for a **UA** (Unnumbered Acknowledgment) from the receiver, which signifies readiness for data transmission. If the sender does not receive the expected **UA** acknowledgment, it can resend the **SET** frame up to a maximum number of attempts. If acknowledgment is not received, the connection establishment will be deemed unsuccessful.

Once data transfer is complete, the connection is terminated with the **DISC** (disconnect) command. The transmitter sends a **DISC** to the receiver, which acknowledges it with a **DISC** response. If the transmitter does not receive this response, it will retry sending the **DISC** command until successful or until the maximum number of attempts is reached. Similarly, if the receiver does not get the **UA** acknowledgment for its **DISC** command, it will also retry until successful or until its maximum attempts are reached. Finally, after the **DISC** is acknowledged, the transmitter sends a **UA** to confirm termination. This structured approach ensures reliable communication while minimizing network interruption risks.

Frame Transmission

When transmitting data frames, the *send_data_frame* function constructs a data frame with a start flag, transmitter address, frame type, calculated BCC1, and end flag. It performs byte stuffing for special values and tracks the number of frames sent.

Overall, the Logical Link Protocol establishes reliable data communication by effectively managing the connection lifecycle, data transmission integrity, and error handling mechanisms, ensuring robust communication over serial connections. The use of a state machine to process incoming frames enhances the protocol's ability to maintain data

integrity and manage errors dynamically, ensuring accurate identification of error sources and successful data recovery.

Application Protocol

The application protocol is designed to facilitate the reliable transfer of files between a transmitter and a receiver over a communication link. It achieves this through a structured approach that encompasses control packet handling, data packet processing, and overall application layer logic.

Control Packet Handling

- **send_control_packet**: This function is responsible for creating control packets that encapsulate important metadata about the file being transferred. The metadata includes the file name and size, formatted using a Type-Length-Value (TLV) structure. Control packets are crucial for signaling the start and end of the file transfer process, allowing the receiver to validate the file's details before proceeding.
- **read_control_packet**: This function reads incoming control packets and extracts critical information, such as the file name and size. The extracted metadata is essential for the receiver to verify the integrity and completeness of the received file.

Data Packet Processing

- **send_data_packets**: This function manages the reading of the file in chunks and encapsulates each chunk into data packets. Each data packet includes a sequence number and encoded length fields to ensure ordered and reliable data transmission. The function also handles retransmissions in the event of packet loss, based on the sequence numbers.
- **read_data_packets**: This function processes incoming data packets. It matches each packet to the expected sequence number to ensure that all packets are received in the correct order. If packets arrive out of order, they are buffered until the expected sequence is received, ensuring the entire file is written correctly to the output file.

Application Layer Logic

- **applicationLayer**: This is the main function that orchestrates the file transfer process. It establishes a connection using the *llopen* function based on the specified role (either transmitter or receiver). In transmitter mode, it calls *send_control_packet* and *send_data_packets* to send the file. In receiver mode, it invokes *read_control_packet* and *read_data_packets* to receive and process the file data. The function concludes by calling *llclose* to gracefully terminate the connection, ensuring a clean end to the communication.

Validation

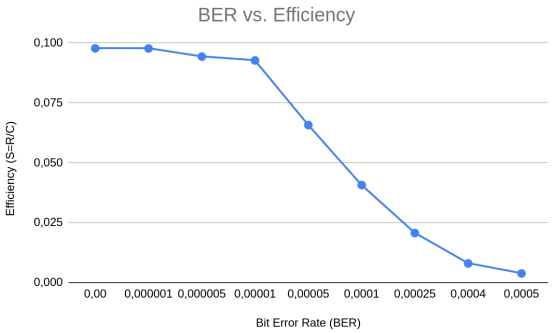
A series of tests were conducted to evaluate the robustness of the program across various scenarios:

- **File Transfers**: Different types of files were transmitted (e.g., *penguin.gif*) to assess handling capabilities.
- **Connection Interruptions**: The connection was deliberately interrupted during operations such as *llopen*, *llwrite*, and *llread* to observe the program's response.
- **Noise Interference**: Tests were performed to simulate noise during transmission, highlighting the system's resilience. However, when noise levels were excessive, some frames with errors were mistakenly accepted as correct due to the limitations in the performance of the error detection techniques (BCC1 and BCC2).
- **Packet Duplication Handling**: The program's ability to manage duplicate packets was tested.
- **File Size Variability**: Files of different sizes were transmitted successfully, with consistent performance observed across all file sizes.

Overall, all tests confirmed that the program performed as expected, except in cases involving excessive noise, where error detection efficacy declined.

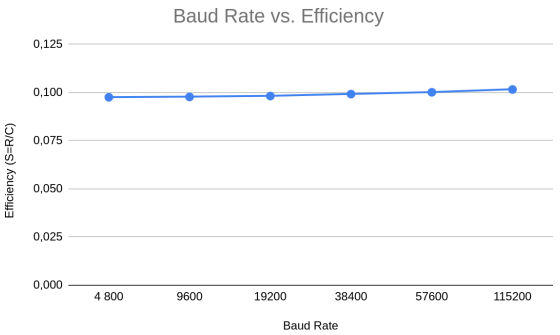
Data Link Protocol Efficiency

The efficiency of the data link protocol is evaluated through tests that measure performance under various conditions. Results are based on the average of five trials. The penguin.gif image file was used to maintain a consistent number of frames across trials, with a maximum payload size of 1000 bytes, except for tests on variations of this value.



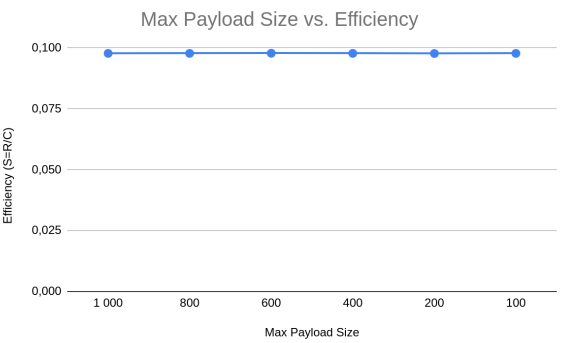
Impact of Bit Error Rate (BER)

Graph 1 shows that BER significantly influences protocol efficiency. As errors increase, the frequency of retransmissions rises, leading to longer execution times and reduced efficiency.



Baud Rate Variation

Graph 2 demonstrates that efficiency remains fairly constant across varying baud rates. While higher baud rates allow faster data transmission, they have a limited effect on overall efficiency, suggesting the protocol maintains stable performance across different transmission speeds.

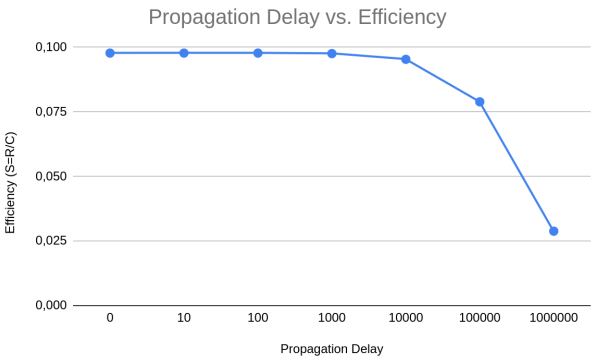


Payload Size Variation

Graph 3 reveals that efficiency is largely unaffected by changes in the maximum payload size. Although larger frames theoretically reduce overhead, the protocol's efficiency remains stable, indicating that payload size variations have minimal impact on overall performance.

Propagation Delay Variation

Graph 4 shows that increasing propagation delay results in lower efficiency due to longer idle times. Extended delays between frame transmission and reception impact performance significantly at higher values



Theoretical Perspective on the Stop-and-Wait Protocol

The Stop-and-Wait protocol requires the sender to wait for an acknowledgment (ACK) after sending a frame. If the frame is received without errors, the receiver sends an ACK; otherwise, it issues a Negative Acknowledgment (NACK). A timeout mechanism is crucial to handle lost frames or acknowledgments, prompting the sender to resend if needed.

I-frames and ACKs use alternating sequence numbers (0 or 1) to distinguish between new and duplicate frames. In our implementation, I-frames are labeled with 0 or 1, while the receiver's responses are designated as RR (ACK) or REJ (NACK) along with their respective sequence numbers.

Frame Transmission and Acknowledgment Process

- **I-Frames with No Errors in Header/Data:**
  - **New Frame:** Data is processed and acknowledged (Ns == 0 ? RR1 : RR0).
  - **Duplicate Frame:** Data is discarded, but acknowledged (Ns == 0 ? RR0 : RR1).

- **I-Frames with Header OK but Data Error (BCC):**
  - **New Frame:** Data discarded; retransmission requested (Ns == 0 ? REJ0 : REJ1).
  - **Duplicate Frame:** Data discarded, acknowledged (Ns == 0 ? RR0 : RR1).

Conclusions

This project successfully implements a Stop-and-Wait ARQ protocol, ensuring reliable point-to-point communication despite data corruption and noise. Its modular design, with components like the state machine and alarm system, effectively maintains data integrity.

While the Stop-and-Wait approach provides reliability, it faces efficiency limitations, especially at lower baud rates or high Frame Error Rates. Advanced protocols like Sliding Window ARQ could enhance performance by allowing multiple frames in transit.

Overall, this project achieves its goals, highlighting the importance of understanding ARQ protocol trade-offs and laying the groundwork for future improvements.

Appendix I - Source Code

application_layer.h

```h
// 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

```c
// Application layer protocol implementation

#include "application_layer.h"
#include "link_layer.h"
#include <stdio.h>
#include <string.h>

#define START 1
#define DATA 2
#define END 3
#define FILE_SIZE 0
#define FILE_NAME 1
#define MAX_SEQUENCE_NUMBER 99
#define MAX_PACKETS (MAX_SEQUENCE_NUMBER + 1)

// Structure for a data packet
typedef struct
{
    int sequence_number;                    // Sequence number of the packet
    unsigned char data[MAX_PAYLOAD_SIZE - 4]; // Data payload
    int data_size;                          // Size of the data
} Packet;

// Function declarations
void initialize_packet_buffer(Packet *packet_buffer);
void process_packet(Packet *packet, FILE *output_file);
int send_control_packet(int control_byte, const char *filename, int file_size);
int handle_receiver(const char *filename);
int handle_transmitter(const char *filename);
int read_control_packet(int *file_size, unsigned char *received_filename);
int read_data_packets(FILE *output_file, const unsigned char *received_filename, int file_size);
int send_data_packets(FILE *file, const char *filename, int file_size);
int check_end_packet(unsigned char *control_packet, int data_size, const unsigned char *filename, int file_size);
```

```c
// Main application layer function
void applicationLayer(const char *serialPort, const char *role, int baudRate,
                      int nTries, int timeout, const char *filename)
{
    LinkLayer connection_parameters;

    // Initialize connection parameters
    strcpy(connection_parameters.serialPort, serialPort);
    connection_parameters.baudRate = baudRate;
    connection_parameters.nRetransmissions = nTries;
    connection_parameters.timeout = timeout;

    // Set the communication role (receiver or transmitter)
    if (strcmp(role, "rx") == 0)
    {
        connection_parameters.role = LlRx; // Receiver role
    }
    else if (strcmp(role, "tx") == 0)
    {
        connection_parameters.role = LlTx; // Transmitter role
    }
    else
    {
        printf("Invalid role: %s. Use 'rx' or 'tx'.\n", role);
        return;
    }

    // Establish a connection
    if (llopen(connection_parameters) < 0)
    {
        printf("Failed to open connection.\n");
        return;
    }

    // Data Transfer
    if (connection_parameters.role == LlRx) // Receiver
    {
        if (handle_receiver(filename) < 0)
        {
            printf("An error occurred during data transfer.\n");
        }
    }
    else if (connection_parameters.role == LlTx) // Transmitter
    {
        if (handle_transmitter(filename) < 0)
        {
            printf("An error occurred during data transfer.\n");
        }
    }

    // Termination
    if (llclose(1) < 0)
    {
        printf("Failed to close connection.\n");
    }
}

////////////////////////////////////////////////
// HELPER FUNCTIONS
////////////////////////////////////////////////

// Initialize packet buffer to mark packets as empty
void initialize_packet_buffer(Packet *packet_buffer)
{
    for (int i = 0; i < MAX_PACKETS; i++)
    {
        packet_buffer[i].sequence_number = -1; // Mark as empty
        packet_buffer[i].data_size = 0;
    }
}

// Process and write the received packet to the output file
void process_packet(Packet *packet, FILE *output_file)
{
    fwrite(packet->data, 1, packet->data_size, output_file); // Write data to file
}

// Send a control packet with file information
int send_control_packet(int control_byte, const char *filename, int file_size)
{
    unsigned char packet[MAX_PAYLOAD_SIZE];
    int packet_size = 0;

    packet[packet_size++] = control_byte; // Control field

    // Include file size in the packet
    packet[packet_size++] = FILE_SIZE;                                      // T
    packet[packet_size++] = (unsigned char)sizeof(file_size);               // L (Length of V)
    memcpy(&packet[packet_size], (unsigned char *)&file_size, sizeof(file_size)); // V
    packet_size += sizeof(file_size);

    // Include file name in the packet
    packet[packet_size++] = FILE_NAME;                      // T
    packet[packet_size++] = (unsigned char)strlen(filename);  // L (Length of V)
    memcpy(&packet[packet_size], filename, strlen(filename)); // V
    packet_size += strlen(filename);

    int bytes_written = llwrite(packet, packet_size);
    if (bytes_written < 0)
    {
        printf("Failed to send control packet.\n");
        return -1;
    }

```

```c

    return 1; // Successfully sent control packet
}

// Read a control packet and extract file information
int read_control_packet(int *file_size, unsigned char *received_filename)
{
    unsigned char control_packet[MAX_PAYLOAD_SIZE];
    int bytes_read = llread(control_packet); // Read the start control packet

    if (bytes_read < 0)
    {
        printf("Failed to read start control packet.\n");
        return -1;
    }

    // Process the start control packet
    if (control_packet[0] == START)
    {
        int index = 1;

        // Extract file size and filename from the control packet
        while (index < bytes_read)
        {
            unsigned char type = control_packet[index++];
            unsigned char length = control_packet[index++];
            if (type == FILE_SIZE)
            {
                memcpy(file_size, &control_packet[index], length); // Get file size
                index += length;
            }
            else if (type == FILE_NAME)
            {
                memcpy(received_filename, &control_packet[index], length); // Get filename
                received_filename[length] = '\0';                          // Null-terminate the string
                index += length;
            }
        }

        return 1; // Successfully read control packet
    }

    printf("Unexpected control packet received: %d\n", control_packet[0]);
    return -1; // Unexpected control packet received
}

// Read incoming data packets and write them to the output file
int read_data_packets(FILE *output_file, const unsigned char *received_filename, int file_size)
{
    // Buffer for out-of-sequence packets
    Packet packet_buffer[MAX_PACKETS];
    initialize_packet_buffer(packet_buffer); // Initialize packet buffer

    unsigned char data_packet[MAX_PAYLOAD_SIZE];
    int expected_sequence_number = 0; // Initialize expected sequence number

    while (1)
    {
        int bytes_read = llread(data_packet); // Read data packet

        if (bytes_read < 0)
        {
            printf("Failed to read data packet.\n");
            return -1; // Error reading data packet
        }

        // Check for end packet
        if (data_packet[0] == END)
        {
            if (check_end_packet(data_packet, bytes_read, received_filename, file_size) < 0)
            {
                printf("Start and End control packet mismatch!\n");
            }
            return 1; // End packet received
        }

        if (data_packet[0] == DATA)
        {
            Packet packet;
            packet.sequence_number = data_packet[1];                          // Get sequence number
            packet.data_size = 256 * (int)data_packet[2] + (int)data_packet[3]; // K = 256 * L2 + L1
            memcpy(packet.data, &data_packet[4], packet.data_size);           // Copy data into packet

            if (packet.data_size + 4 != bytes_read)
            {
                // If link_layer is correct, this will never happen
                printf("Packet size mismatch, expected %d bytes but got %d.\n", packet.data_size + 4, bytes_read);
                packet.data_size = bytes_read - 4; // Adjust data size if mismatch
            }

            if (packet.sequence_number == expected_sequence_number)
            {
                process_packet(&packet, output_file);                                 // Process the correctly sequenced packet
                expected_sequence_number = (expected_sequence_number + 1) % MAX_PACKETS; // Update expected sequence number

                // Process any buffered packets that are now in order
                while (packet_buffer[expected_sequence_number].sequence_number != -1)
                {
                    process_packet(&packet_buffer[expected_sequence_number], output_file); // Process buffered packet

                    // Clear the processed packet
                    packet_buffer[expected_sequence_number].sequence_number = -1; // Mark as empty
                    packet_buffer[expected_sequence_number].data_size = 0;

                    expected_sequence_number = (expected_sequence_number + 1) % MAX_PACKETS; // Update expected number
```

```
241
242                     expected_sequence_number = (expected_sequence_number + 1) % MAX_PACKETS; // Update expected number
243                 }
244             }
245             else if (packet_buffer[packet.sequence_number].sequence_number == -1)
246             {
247                 // Store out-of-sequence packets in the buffer
248                 packet_buffer[packet.sequence_number] = packet; // Buffer the packet
249             }
250         }
251     }
252
253     return 1; // Successfully processed all packets
254 }
255
256 // Handle the receiver role and manage the file reception
257 int handle_receiver(const char *filename)
258 {
259     int file_size = 0;
260     unsigned char received_filename[MAX_PAYLOAD_SIZE];
261
262     // Read the control packet to get file info
263     if (read_control_packet(&file_size, received_filename) < 0)
264     {
265         return -1; // Error reading control packet
266     }
267
268     // Open file for writing
269     FILE *output_file = fopen(filename, "wb");
270     if (!output_file)
271     {
272         printf("Error opening file for writing: %s\n", filename);
273         return -1; // Error opening output file
274     }
275
276     // Read and process data packets
277     if (read_data_packets(output_file, received_filename, file_size) < 0)
278     {
279         fclose(output_file);
280         return -1; // Error processing data packets
281     }
282
283     fclose(output_file); // Close output file
284     return 1;            // Successfully received file
285 }
286
287 // Send data packets from the file
288 int send_data_packets(FILE *file, const char *filename, int file_size)
289 {
290     unsigned char packet[MAX_PAYLOAD_SIZE];
291     int bytes_read;
292     int sequence_number = 0; // Initialize sequence number
293
294     while ((bytes_read = fread(packet, 1, MAX_PAYLOAD_SIZE - 4, file)) > 0)
295     {
296         unsigned char data_packet[MAX_PAYLOAD_SIZE];
297         int packet_size = 0;
298
299         // Construct data packet
300         data_packet[packet_size++] = DATA;            // Packet type
301         data_packet[packet_size++] = sequence_number; // Add sequence number
302
303         // K = 256 * L2 + L1
304         int L1 = bytes_read % 256; // Low byte
305         int L2 = bytes_read / 256; // High byte
306
307         data_packet[packet_size++] = (unsigned char)L2; // Add high byte
308         data_packet[packet_size++] = (unsigned char)L1; // Add low byte
309
310         memcpy(&data_packet[packet_size], packet, bytes_read); // Add data to packet
311         packet_size += bytes_read;
312
313         if (llwrite(data_packet, packet_size) < 0)
314         {
315             printf("Failed to send data packet.\n");
316             return -1; // Error sending data packet
317         }
318
319         sequence_number = (sequence_number + 1) % (MAX_SEQUENCE_NUMBER + 1); // Update sequence number
320     }
321
322     return 1; // Successfully sent all data packets
323 }
324
325 // Handle the transmitter role and manage the file transmission
326 int handle_transmitter(const char *filename)
327 {
328     FILE *file = fopen(filename, "rb"); // Open file for reading
329     if (!file)
330     {
331         printf("Error opening file: %s\n", filename);
332         return -1; // Error opening input file
333     }
334
335     fseek(file, 0, SEEK_END);    // Move to the end of the file
336     int file_size = ftell(file); // Get the file size
337     fseek(file, 0, SEEK_SET);    // Move back to the start of the file
338
339     // Send start control packet with file info
340     if (send_control_packet(START, filename, file_size) < 0)
341     {
342         fclose(file);
343         return -1; // Error sending start control packet
344     }
```

```
345
346     // Send data packets
347     if (send_data_packets(file, filename, file_size) < 0)
348     {
349         fclose(file);
350         return -1; // Error sending data packets
351     }
352
353     // Send end control packet
354     if (send_control_packet(END, filename, file_size) < 0)
355     {
356         fclose(file);
357         return -1; // Error sending end control packet
358     };
359
360     fclose(file); // Close input file
361     return 1;     // Successfully sent file
362 }
363
364 // Check the end packet for consistency with the start packet
365 int check_end_packet(unsigned char *control_packet, int data_size, const unsigned char *filename, int file_size)
366 {
367     int index = 1; // Start from first data after control byte
368
369     int received_file_size = 0; // To store received file size
370     unsigned char received_filename[MAX_PAYLOAD_SIZE];
371
372     int end_filename_sent = 0; // Flag to check if end filename was sent
373
374     // Extract file size and filename from the control packet
375     while (index < data_size)
376     {
377         unsigned char type = control_packet[index++];
378         unsigned char length = control_packet[index++];
379         if (type == FILE_SIZE)
380         {
381             memcpy(&received_file_size, &control_packet[index], length); // Get received file size
382             index += length;
383
384             if (received_file_size != file_size)
385             {
386                 return -1; // File size mismatch
387             }
388         }
389         else if (type == FILE_NAME)
390         {
391             memcpy(received_filename, &control_packet[index], length); // Get received filename
392             received_filename[length] = '\0';                          // Null-terminate
393             index += length;
394             end_filename_sent = 1; // Mark filename sent
395
396             if (strcmp((const char *)received_filename, (const char *)filename) != 0)
397             {
398                 return -1; // Filename mismatch
399             }
400         }
401     }
402
403     return (strlen((const char *)filename) > 0 && end_filename_sent) ? 1 : -1;
404 }
405
```

link_layer.h

```c
// 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);

// 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 "state_machine.h"
#include "alarm.h"
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

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

// Global variables
LinkLayer connection_parameters; // Connection parameters for link layer
int frame_number = 0;            // Current frame number (0 or 1)
int frames_received = 0;         // Count of frames received

// Statistics structure
extern struct ll_statistics statistics;

// Helper Functions prototypes
int safe_write(const unsigned char *bytes, int num_bytes);
int send_SET();
int send_ACK();
int llopen_receiver();
int llopen_transmitter();
int send_data_frame(const unsigned char *buf, int buf_size);
int send_RR();
int send_REJ();
int send_DISC();
int llclose_receiver();
int llclose_transmitter();
void show_statistics(struct ll_statistics statistics);

////////////////////////////////////////////////
// LLOPEN
////////////////////////////////////////////////
int llopen(LinkLayer connectionParameters)
{
    // Open the serial port with specified parameters
    if (openSerialPort(connectionParameters.serialPort,
                       connectionParameters.baudRate) < 0)
    {
        return -1; // Error opening serial port
    }

    connection_parameters = connectionParameters; // Store connection parameters

    // Handle connection based on role (Receiver or Transmitter)
    switch (connectionParameters.role)
    {
    case LlRx: // Receiver
        if (llopen_receiver() < 0)
            return -1; // Error during receiver connection
        break;
    case LlTx: // Transmitter
        if (llopen_transmitter() < 0)
            return -1; // Error during transmitter connection
        break;
    default:
        return -1; // Invalid role
    }

    return 1; // Connection successful
}

////////////////////////////////////////////////
// LLWRITE
////////////////////////////////////////////////
int llwrite(const unsigned char *buf, int bufSize)
{
    struct state_machine machine;
    // Create a type WRITE state machine
    create_state_machine(&machine, WRITE, (frame_number == 0 ? RR1 : RR0), REPLY_FROM_RECEIVER_ADDRESS, START);

    unsigned int attempt = 0;
    extern int alarm_enabled;
    extern int alarm_count;
    alarm_enabled = FALSE; // Disable alarm initially
    alarm_count = 0;       // Reset alarm count

    (void)signal(SIGALRM, alarm_handler); // Set signal handler for alarm

    // Retry sending data frame based on the number of retransmissions
    while (attempt < connection_parameters.nRetransmissions)
    {
        attempt++;

        // Attempt to send the data frame
        if (send_data_frame(buf, bufSize) < 0) // Failed to send frame
        {
            alarm(0);                          // Disable alarm on failure
            statistics.num_retransmissions++; // Count retransmission
```

```
            statistics.num_retransmissions++; // Count retransmission
            continue;                         // Retry sending frame
        }

        alarm(connection_parameters.timeout); // Set alarm for timeout
        alarm_enabled = TRUE;                 // Enable alarm
        machine.state = START;                // Reset state machine

        // Wait for response while alarm is enabled
        while (alarm_enabled)
        {
            unsigned char byte = 0;
            int read_byte = readByteSerialPort(&byte); // Read byte from serial port

            if (read_byte == 0)
            {
                continue; // No bytes read, continue waiting
            }
            else if (read_byte < 0)
            {
                printf("Read ERROR!"); // Error reading byte
                return -1;
            }

            // Process the byte through the state machine
            state_machine(&machine, byte);
            if (machine.state == STP && machine.REJ) // REJ received
            {
                statistics.num_REJ_received++; // Count REJ received
                statistics.num_timeouts--;     // No timeout when REJ is received
                alarm(0);                      // Disable alarm
                alarm_enabled = FALSE;         // Disable alarm
                attempt--;                     // Stay on the same attempt
                break;                         // Send frame again
            }
            else if (machine.state == STP) // RR received
            {
                frame_number = 1 - frame_number; // Switch frame number
                statistics.num_RR_received++;    // Count RR received
                alarm(0);                        // Disable alarm
                alarm_enabled = FALSE;           // Disable alarm
                return bufSize;                  // Return size of buffer written
            }
        }
        statistics.num_retransmissions++; // Increment retransmission count
        statistics.num_timeouts++;        // Increment timeout count
    }
    printf("Failed to send frame after %d attempts\n", connection_parameters.nRetransmissions);
    return -1; // Failed to send frame after retries
}

////////////////////////////////////////////////
// LLREAD
////////////////////////////////////////////////
int llread(unsigned char *packet)
{
    struct state_machine machine;
    // Create a type READ state machine
    create_state_machine(&machine, READ, (frame_number == 0 ? I_FRAME_0 : I_FRAME_1), TRANSMITTER_ADDRESS, START);

    do
    {
        unsigned char byte = 0;
        int read_byte = readByteSerialPort(&byte); // Read byte from serial port

        if (read_byte == 0)
        {
            continue; // No bytes read, continue waiting
        }
        else if (read_byte < 0)
        {
            printf("Read ERROR!"); // Error reading byte
            return -1;
        }

        // Process the byte through the state machine
        state_machine(&machine, byte);

        // Handle received frames based on state machine state
        if (machine.state == STP && machine.ACK && frames_received == 0) // SET received
        {
            statistics.num_SET_received++; // Count SET received

            if (send_ACK() < 0) // Send ACK command
                return -1;      // Error sending ACK

            machine.state = START; // Reset state for next frame
        }
        else if (machine.state == STP && machine.duplicate) // Duplicate received
        {
            statistics.num_I_frames_received++; // Count I frames received
            statistics.num_duplicated_frames++; // Count duplicated frames

            if (send_RR() < 0)     // Send RR command
                return -1;         // Error sending RR
            machine.state = START; // Reset state for next frame
        }
        else if (machine.state == STP && machine.REJ) // New frame with bad data received
        {
            statistics.num_I_frames_received++; // Count I frames received
```

```
            machine.state = START; // Reset state for next frame
        }
        else if (machine.state == STP && machine.REJ) // New frame with bad data received
        {
            statistics.num_I_frames_received++; // Count I frames received

            if (send_REJ() < 0)    // Send REJ command
                return -1;         // Error sending REJ
            machine.state = START; // Reset state for next frame
        }
        else if (machine.state == STP) // Frame received
        {
            statistics.num_I_frames_received++; // Count I frames received
            frames_received++;                  // Increment frames received count
            break;                              // Frame received successfully
        }
    } while (machine.state != STP);

    frame_number = 1 - frame_number; // Switch frame number
    if (send_RR() < 0)               // Send RR command
        return -1;                   // Error sending RR

    memcpy(packet, machine.buf, machine.buf_size); // Copy received packet to provided buffer

    return machine.buf_size; // Return size of received packet
}

////////////////////////////////////////////
// LLCLOSE
////////////////////////////////////////////
int llclose(int showStatistics)
{
    int clstat = 1; // Connection status

    // Handle connection closure based on role
    if (connection_parameters.role == LlRx)
    {
        if (llclose_receiver() < 0)
            clstat = -1; // Error during receiver close
    }
    else if (connection_parameters.role == LlTx)
    {
        if (llclose_transmitter() < 0)
            clstat = -1; // Error during transmitter close
    }

    // Close the serial port
    if (closeSerialPort() < 0)
        clstat = -1; // Error closing serial port

    // Show statistics if requested
    if (showStatistics)
        show_statistics(statistics);

    return clstat; // Return connection status
}

////////////////////////////////////////////
// HELPER FUNCTIONS
////////////////////////////////////////////

// Safely writes bytes to the serial port, handling partial writes
int safe_write(const unsigned char *bytes, int num_bytes)
{
    int total_bytes_written = 0;

    // Write bytes until all requested bytes are written
    while (total_bytes_written < num_bytes)
    {
        int bytes_to_write = num_bytes - total_bytes_written;
        int bytes_written = writeBytesSerialPort(bytes + total_bytes_written, bytes_to_write);

        if (bytes_written < 0)
        {
            printf("ERROR writing to serial port!\n");
            return -1; // Indicate an error occurred
        }

        total_bytes_written += bytes_written; // Update total written
    }

    return total_bytes_written; // Return total bytes written
}

// Send SET command to establish connection
int send_SET()
{
    unsigned char buf[5] = {FLAG, TRANSMITTER_ADDRESS, SET, 0, FLAG};
    buf[3] = buf[1] ^ buf[2]; // Calculate BCC1

    if (safe_write(buf, 5) < 0)
    {
        printf("Failed to send SET command.\n");
        return -1; // Error sending SET command
    }

    statistics.num_SET_sent++; // Count SET command sent
    return 1;                  // Successful send
}
```

```c
            }

            state_machine(&machine, byte); // Process the read byte with the state machine

            // Check if the state machine has reached the STOP state (STP)
            if (machine.state == STP)
            {
                alarm(0);                    // Disable the alarm
                alarm_enabled = FALSE;       // Mark alarm as disabled
                statistics.num_UA_received++; // Increment the count of UA frames received
                return 1;                    // Successful connection establishment
            }
        }
        statistics.num_retransmissions++; // Increment retransmission count
        statistics.num_timeouts++;        // Increment timeout count
    }
    printf("Failed to establish connection after %d attempts\n", connection_parameters.nRetransmissions);
    return -1; // Return error if maximum retransmissions are reached without success
}

// Function to send a data frame over the serial connection
int send_data_frame(const unsigned char *buf, int buf_size)
{
    // Allocate memory for the frame:
    // (data_size + BCC2) * 2 for potential byte stuffing + (F; A; C; BCC1) + F
    unsigned char frame[(buf_size + 1) * 2 + 5];
    int frame_size = 0; // Initialize frame size counter

    // Start constructing the frame
    frame[frame_size++] = FLAG;                                     // Start flag
    frame[frame_size++] = TRANSMITTER_ADDRESS;                      // Transmitter address
    frame[frame_size++] = (frame_number == 0) ? I_FRAME_0 : I_FRAME_1; // Frame type (I_FRAME_0 or I_FRAME_1)
    frame[frame_size++] = frame[1] ^ frame[2];                      // Calculate BCC1 (XOR of address and control field)

    unsigned char BCC2 = 0; // Initialize BCC2
    for (int i = 0; i < buf_size; i++)
    {
        if (buf[i] == FLAG) // Check for FLAG byte to perform byte stuffing
        {
            frame[frame_size++] = ESC;      // Add ESC before FLAG
            frame[frame_size++] = ESC_FLAG; // Escape FLAG
        }
        else if (buf[i] == ESC) // Check for ESC byte to perform byte stuffing
        {
            frame[frame_size++] = ESC;     // Add ESC before ESC
            frame[frame_size++] = ESC_ESC; // Escape ESC
        }
        else
        {
            frame[frame_size++] = buf[i]; // Add data byte to frame
        }
        BCC2 ^= buf[i]; // Compute BCC2 using XOR
    }

    // Byte Stuff BCC2
    if (BCC2 == FLAG) // Check if BCC2 is equal to the FLAG byte
    {
        frame[frame_size++] = ESC;      // Add ESC before BCC2
        frame[frame_size++] = ESC_FLAG; // Escape FLAG
    }
    else if (BCC2 == ESC) // Check if BCC2 is equal to the ESC byte
    {
        frame[frame_size++] = ESC;     // Add ESC before BCC2
        frame[frame_size++] = ESC_ESC; // Escape ESC
    }
    else
    {
        frame[frame_size++] = BCC2; // Add BCC2 to frame
    }

    frame[frame_size++] = FLAG; // End flag

    // Attempt to write the frame to the serial port
    if (safe_write(frame, frame_size) < 0)
    {
        printf("Failed to send frame %d!\n", frame_number);
        return -1; // Return -1 on failure
    }

    statistics.num_I_frames_sent++; // Increment the count of I frames sent
    return 1;                       // Return 1 on success
}

// Function to send a RR command
int send_RR()
{
    // Create a buffer to hold the RR frame
    unsigned char buf[5] = {FLAG, REPLY_FROM_RECEIVER_ADDRESS, frame_number == 0 ? RR0 : RR1, 0, FLAG};
    buf[3] = buf[1] ^ buf[2]; // Calculate BCC1

    // Attempt to send the RR command
    if (safe_write(buf, 5) < 0)
    {
        printf("Failed to send RR%d command.\n", frame_number);
        return -1; // Return -1 on failure
    }

    statistics.num_RR_sent++; // Increment the count of RR commands sent
    return 1;                 // Return 1 on success
}
```

```c
// Send ACK command to acknowledge receipt
int send_ACK()
{
    unsigned char buf[5] = {FLAG, 0, UA, 0, FLAG};
    if (connection_parameters.role == LlRx)
    {
        buf[1] = REPLY_FROM_RECEIVER_ADDRESS; // Set address for receiver case
    }
    else if (connection_parameters.role == LlTx)
    {
        buf[1] = REPLY_FROM_TRANSMITTER_ADDRESS; // Set address for trasmitter case
    }
    buf[3] = buf[1] ^ buf[2]; // Calculate BCC1

    if (safe_write(buf, 5) < 0)
    {
        printf("Failed to send ACK command.\n");
        return -1; // Error sending UA command
    }

    statistics.num_UA_sent++; // Count UA command sent
    return 1;                 // Successful send
}

// Function to establish a connection on the receiver side
int llopen_receiver()
{
    // Initialize the state machine for connection establishment
    struct state_machine machine;
    create_state_machine(&machine, CONNECTION, SET, TRANSMITTER_ADDRESS, START);

    // Loop until the state machine reaches the STOP state (STP)
    do
    {
        unsigned char byte = 0;                   // Variable to store the byte read from the serial port
        int read_byte = readByteSerialPort(&byte); // Read a byte from the serial port

        if (read_byte == 0)
        {
            continue; // No bytes read, continue waiting
        }
        else if (read_byte < 0)
        {
            printf("Read ERROR!"); // Error reading byte
            return -1;
        }

        state_machine(&machine, byte); // Process the read byte with the state machine

    } while (machine.state != STP);

    statistics.num_SET_received++; // Increment the count of SET frames received
    return send_ACK();             // Send an acknowledgment (ACK) back to the transmitter
}

// Function to establish a connection on the transmitter side
int llopen_transmitter()
{
    // Initialize the state machine for connection establishment
    struct state_machine machine;
    create_state_machine(&machine, CONNECTION, UA, REPLY_FROM_RECEIVER_ADDRESS, START);

    unsigned int attempt = 0;             // Counter for connection attempts
    extern int alarm_enabled;             // Flag indicating if the alarm is enabled
    extern int alarm_count;               // Counter for alarm events
    alarm_enabled = FALSE;                // Initially, alarm is disabled
    alarm_count = 0;                      // Reset alarm count
    (void)signal(SIGALRM, alarm_handler); // Set the signal handler for alarms

    // Loop until the maximum number of retransmissions is reached
    while (attempt < connection_parameters.nRetransmissions)
    {
        attempt++; // Increment the attempt counter

        if (send_SET() < 0) // Attempt to send the SET frame
        {
            alarm(0);                         // Disable the alarm
            statistics.num_retransmissions++; // Increment retransmission count
            continue;                         // Retry sending SET if it fails
        }

        alarm(connection_parameters.timeout); // Set the alarm for timeout duration
        alarm_enabled = TRUE;                 // Enable the alarm
        machine.state = START;                // Reset the state machine state

        // Loop while the alarm is enabled (waiting for UA frame)
        while (alarm_enabled)
        {
            unsigned char byte = 0;                   // Variable to store the byte read from the serial port
            int read_byte = readByteSerialPort(&byte); // Read a byte from the serial port

            if (read_byte == 0)
            {
                continue; // No bytes read, continue waiting
            }
            else if (read_byte < 0)
            {
                printf("Read ERROR!"); // Error reading byte
                return -1;
```

```c

// Function to send a REJ command
int send_REJ()
{
    // Create a buffer to hold the REJ frame
    unsigned char buf[5] = {FLAG, REPLY_FROM_RECEIVER_ADDRESS, frame_number == 0 ? REJ0 : REJ1, 0, FLAG};
    buf[3] = buf[1] ^ buf[2]; // Calculate BCC1

    // Attempt to send the REJ command
    if (safe_write(buf, 5) < 0)
    {
        printf("Failed to send REJ%d command.\n", frame_number);
        return -1; // Return -1 on failure
    }

    statistics.num_REJ_sent++; // Increment the count of REJ commands sent
    return 1;                  // Return 1 on success
}

// Function to send a DISC command
int send_DISC()
{
    // Create a buffer to hold the DISC frame
    unsigned char buf[5] = {FLAG, 0, DISC, 0, FLAG};

    // Set the second byte based on the role (Receiver or Transmitter)
    if (connection_parameters.role == LlRx)
    {
        buf[1] = RECEIVER_ADDRESS; // Receiver's address
    }
    else if (connection_parameters.role == LlTx)
    {
        buf[1] = TRANSMITTER_ADDRESS; // Transmitter's address
    }

    buf[3] = buf[1] ^ buf[2]; // Calculate BCC1

    // Attempt to send the DISC command
    if (safe_write(buf, 5) < 0)
    {
        printf("Failed to send DISC command.\n");
        return -1; // Return -1 on failure
    }

    statistics.num_DISC_sent++; // Increment the count of DISC commands sent
    return 1;                   // Return 1 on success
}

// Function to close the connection from the transmitter's side
int llclose_transmitter()
{
    // Initialize the state machine for the disconnection process
    struct state_machine machine;
    create_state_machine(&machine, DISCONNECTION, DISC, RECEIVER_ADDRESS, START);

    unsigned int attempt = 0;
    extern int alarm_enabled;
    extern int alarm_count;
    alarm_enabled = FALSE; // Disable alarm
    alarm_count = 0;       // Reset alarm count

    (void)signal(SIGALRM, alarm_handler); // Set the alarm handler

    // Try sending the DISC command up to nRetransmissions times
    while (attempt < connection_parameters.nRetransmissions)
    {
        attempt++;

        if (send_DISC() < 0) // Attempt to send DISC
        {
            alarm(0);                       // Disable alarm if sending fails
            statistics.num_retransmissions++; // Increment retransmission count
            continue;                       // Retry sending DISC
        }

        alarm(connection_parameters.timeout); // Set the alarm with the timeout value
        alarm_enabled = TRUE;                 // Enable the alarm
        machine.state = START;                // Reset state machine state

        // Wait for a response while the alarm is enabled
        while (alarm_enabled)
        {
            unsigned char byte = 0;
            int read_byte = readByteSerialPort(&byte); // Read a byte from the serial port

            if (read_byte == 0)
            {
                continue; // No bytes read, continue waiting
            }
            else if (read_byte < 0)
            {
                printf("Read ERROR!");
                return -1; // Return error on read failure
            }

            // Process the received byte with the state machine
            state_machine(&machine, byte);

            if (machine.state == STP) // If in STOP state, DISC received successfully
            {
```

```
            {
                statistics.num_DISC_received++; // Increment DISC received count
                alarm(0);                       // Disable alarm
                alarm_enabled = FALSE;          // Disable alarm flag

                if (send_ACK() < 0) // Send ACK in response
                    return -1;

                return 1; // Return success
            }
        }
        statistics.num_retransmissions++; // Increment retransmission count on timeout
        statistics.num_timeouts++;        // Increment timeout count
    }
    printf("Failed to send DISC after %d attempts\n", connection_parameters.nRetransmissions);
    return -1; // Return error after max attempts
}

// Function to close the connection from the receiver's side
int llclose_receiver()
{
    // Initialize the state machine for receiving the DISC frame
    struct state_machine machine;
    create_state_machine(&machine, DISCONNECTION, DISC, TRANSMITTER_ADDRESS, START);

    // Wait for the DISC frame from the transmitter
    do
    {
        unsigned char byte = 0;
        int read_byte = readByteSerialPort(&byte); // Read a byte from the serial port

        if (read_byte == 0)
        {
            continue; // No bytes read, continue waiting
        }
        else if (read_byte < 0)
        {
            printf("Read ERROR!");
            return -1; // Return error on read failure
        }

        // Process the received byte with the state machine
        state_machine(&machine, byte);

        if (machine.state == STP) // If in STOP state, DISC received successfully
        {
            statistics.num_DISC_received++; // Increment DISC received count
            break;                          // Exit the loop
        }
    } while (machine.state != STP);

    // Prepare to reply with a DISC and wait for the UA reply
    create_state_machine(&machine, DISCONNECTION, UA, REPLY_FROM_TRANSMITTER_ADDRESS, START);

    unsigned int attempt = 0;
    extern int alarm_enabled;
    extern int alarm_count;
    alarm_enabled = FALSE; // Disable alarm
    alarm_count = 0;       // Reset alarm count

    (void)signal(SIGALRM, alarm_handler); // Set the alarm handler

    // Try sending the DISC command up to nRetransmissions times
    while (attempt < connection_parameters.nRetransmissions)
    {
        attempt++;

        if (send_DISC() < 0) // Attempt to send DISC
        {
            alarm(0);                       // Disable alarm if sending fails
            statistics.num_retransmissions++; // Increment retransmission count
            continue;                       // Retry sending DISC
        }

        alarm(connection_parameters.timeout); // Set the alarm with the timeout value
        alarm_enabled = TRUE;                 // Enable the alarm
        machine.state = START;                // Reset state machine state

        // Wait for a response while the alarm is enabled
        while (alarm_enabled)
        {
            unsigned char byte = 0;
            int read_byte = readByteSerialPort(&byte); // Read a byte from the serial port

            if (read_byte == 0)
            {
                continue; // No bytes read, continue waiting
            }
            else if (read_byte < 0)
            {
                printf("Read ERROR!");
                return -1; // Return error on read failure
            }

            // Process the received byte with the state machine
            state_machine(&machine, byte);

            if (machine.state == STP) // If in STOP state, UA received successfully
            {
                statistics.num_UA_received++; // Increment UA received count
```

```c
    alarm_count = 0;       // Reset alarm count

    (void)signal(SIGALRM, alarm_handler); // Set the alarm handler

    // Try sending the DISC command up to nRetransmissions times
    while (attempt < connection_parameters.nRetransmissions)
    {
        attempt++;

        if (send_DISC() < 0) // Attempt to send DISC
        {
            alarm(0);                          // Disable alarm if sending fails
            statistics.num_retransmissions++; // Increment retransmission count
            continue;                          // Retry sending DISC
        }

        alarm(connection_parameters.timeout); // Set the alarm with the timeout value
        alarm_enabled = TRUE;                 // Enable the alarm
        machine.state = START;                // Reset state machine state

        // Wait for a response while the alarm is enabled
        while (alarm_enabled)
        {
            unsigned char byte = 0;
            int read_byte = readByteSerialPort(&byte); // Read a byte from the serial port

            if (read_byte == 0)
            {
                continue; // No bytes read, continue waiting
            }
            else if (read_byte < 0)
            {
                printf("Read ERROR!");
                return -1; // Return error on read failure
            }

            // Process the received byte with the state machine
            state_machine(&machine, byte);

            if (machine.state == STP) // If in STOP state, UA received successfully
            {
                statistics.num_UA_received++; // Increment UA received count
                alarm(0);                     // Disable alarm
                alarm_enabled = FALSE;        // Disable alarm flag
                return 1;                     // Return success
            }
        }
        statistics.num_retransmissions++; // Increment retransmission count on timeout
        statistics.num_timeouts++;        // Increment timeout count
    }
    printf("Failed to send DISC after %d attempts\n", connection_parameters.nRetransmissions);
    return -1; // Return error after max attempts
}

// Function to display statistics about the communication
void show_statistics(struct ll_statistics statistics)
{
    // Calculate total frames sent and received
    int num_frames_sent = statistics.num_SET_sent +
                          statistics.num_UA_sent +
                          statistics.num_RR_sent +
                          statistics.num_REJ_sent +
                          statistics.num_I_frames_sent +
                          statistics.num_DISC_sent;

    int num_frames_received = statistics.num_SET_received +
                              statistics.num_UA_received +
                              statistics.num_RR_received +
                              statistics.num_REJ_received +
                              statistics.num_I_frames_received +
                              statistics.num_DISC_received;

    // Print out the statistics in a readable format
    printf("\n");
    printf("Total Frames Sent: %d\n", num_frames_sent);
    printf("Total Frames Received: %d\n", num_frames_received);
    printf("Total SET Frames Sent: %d\n", statistics.num_SET_sent);
    printf("Total SET Frames Received: %d\n", statistics.num_SET_received);
    printf("Total UA Frames Sent: %d\n", statistics.num_UA_sent);
    printf("Total UA Frames Received: %d\n", statistics.num_UA_received);
    printf("Total RR Frames Sent: %d\n", statistics.num_RR_sent);
    printf("Total RR Frames Received: %d\n", statistics.num_RR_received);
    printf("Total REJ Frames Sent: %d\n", statistics.num_REJ_sent);
    printf("Total REJ Frames Received: %d\n", statistics.num_REJ_received);
    printf("Total I Frames Sent: %d\n", statistics.num_I_frames_sent);
    printf("Total I Frames Received: %d\n", statistics.num_I_frames_received);
    printf("Total DISC Frames Sent: %d\n", statistics.num_DISC_sent);
    printf("Total DISC Frames Received: %d\n", statistics.num_DISC_received);
    printf("Total Invalid BCC1 Received: %d\n", statistics.num_invalid_BCC1_received);
    printf("Total Invalid BCC2 Received: %d\n", statistics.num_invalid_BCC2_received);
    printf("Total Duplicated Frames Received: %d\n", statistics.num_duplicated_frames);
    printf("Total Timeouts: %d\n", statistics.num_timeouts);
    printf("Total Retransmissions: %d\n", statistics.num_retransmissions);
    printf("\n");
}
```

state_machine.h

```h
#ifndef _STATE_MACHINE_H_
#define _STATE_MACHINE_H_

#include "link_layer.h"

// Enumeration for the different states of the state machine
enum state_machine_state
{
    START,    // Initial state, waiting for FLAG
    FLAG_RCV, // FLAG received, waiting for Address byte
    A_RCV,    // Address byte received, waiting for Control byte
    C_RCV,    // Control byte received, waiting for BCC1
    BCC1_OK,  // BCC1 validated, waiting for data or FLAG
    STP       // STOP state, end of frame processing
};

// Enumeration for the different types of state machines
enum state_machine_type
{
    CONNECTION,   // State machine for connection establishment
    READ,         // State machine for reading data
    WRITE,        // State machine for writing data
    DISCONNECTION // State machine for disconnection
};

// Structure representing a state machine instance
struct state_machine
{
    enum state_machine_type type;               // Type of the state machine (e.g., READ, WRITE)
    unsigned char control_byte;                 // Control byte for the current frame
    unsigned char address_byte;                 // Address byte for the current frame
    enum state_machine_state state;             // Current state of the state machine
    unsigned char buf[MAX_PAYLOAD_SIZE * 2 + 2]; // Buffer for storing incoming data
    int buf_size;                               // Current size of the buffer
    unsigned char BCC1;                         // BCC1 value for error checking
    unsigned char BCC2;                         // BCC2 value for error checking
    unsigned char escape_sequence;              // Flag to indicate if an escape sequence is in progress
    unsigned char REJ;                          // REJ flag to indicate a rejection
    unsigned char ACK;                          // ACK flag to indicate acknowledgment
    unsigned char duplicate;                    // Flag to indicate a duplicate frame
};

// Structure to hold statistics related to link layer operations
struct ll_statistics
{
    int num_SET_sent;              // Number of SET frames sent
    int num_SET_received;          // Number of SET frames received
    int num_UA_sent;               // Number of UA frames sent
    int num_UA_received;           // Number of UA frames received
    int num_RR_sent;               // Number of RR frames sent
    int num_RR_received;           // Number of RR frames received
    int num_REJ_sent;              // Number of REJ frames sent
    int num_REJ_received;          // Number of REJ frames received
    int num_I_frames_sent;         // Number of I frames sent
    int num_I_frames_received;     // Number of I frames received
    int num_DISC_sent;             // Number of DISC frames sent
    int num_DISC_received;         // Number of DISC frames received
    int num_duplicated_frames;     // Number of duplicated frames detected
    int num_retransmissions;       // Number of retransmissions
    int num_timeouts;              // Number of timeouts
    int num_invalid_BCC1_received; // Number of invalid BCC1 received
    int num_invalid_BCC2_received; // Number of invalid BCC2 received
};

// Constants defining special bytes used in the protocol
#define FLAG 0x7E                           // Flag byte to denote start/end of frame
#define TRANSMITTER_ADDRESS 0x03            // Address of the transmitter
#define RECEIVER_ADDRESS 0x01               // Address of the receiver
#define REPLY_FROM_TRANSMITTER_ADDRESS 0x01 // Expected reply address from the transmitter
#define REPLY_FROM_RECEIVER_ADDRESS 0x03    // Expected reply address from the receiver
#define SET 0x03                            // Control byte for SET frame
#define UA 0x07                             // Control byte for UA frame
#define DISC 0x0B                           // Control byte for DISC frame
#define I_FRAME_0 0x00                      // Control byte for I frame 0
#define I_FRAME_1 0x80                      // Control byte for I frame 1
#define ESC 0x7D                            // Escape byte for byte-stuffing
#define ESC_FLAG 0x5E                       // Transformed FLAG byte after escaping
#define ESC_ESC 0x5D                        // Transformed ESC byte after escaping
#define RR0 0xAA                            // Control byte for RR frame 0
#define RR1 0xAB                            // Control byte for RR frame 1
#define REJ0 0x54                           // Control byte for REJ frame 0
#define REJ1 0x55                           // Control byte for REJ frame 1

// Extern declaration of statistics structure
extern struct ll_statistics statistics;

// Function declarations for state machine operations
void create_state_machine(struct state_machine *machine, enum state_machine_type type, unsigned char control_byte, unsigned char address_byte, enum state_machine_state state);
void process_read_BCC1_OK(struct state_machine *machine, unsigned char byte);
void state_machine(struct state_machine *machine, unsigned char byte);
void state_machine_START(struct state_machine *machine, unsigned char byte);
void state_machine_FLAG_RCV(struct state_machine *machine, unsigned char byte);
void state_machine_A_RCV(struct state_machine *machine, unsigned char byte);
void state_machine_C_RCV(struct state_machine *machine, unsigned char byte);
void state_machine_BCC1_OK(struct state_machine *machine, unsigned char byte);

#endif // _STATE_MACHINE_H_
```

state_machine.c

```c
#include "state_machine.h"
#include <stdio.h>
#include <string.h>

// Structure to hold statistics for various control frames and events
struct ll_statistics statistics = {
    .num_SET_sent = 0,
    .num_SET_received = 0,
    .num_UA_sent = 0,
    .num_UA_received = 0,
    .num_RR_sent = 0,
    .num_RR_received = 0,
    .num_REJ_sent = 0,
    .num_REJ_received = 0,
    .num_I_frames_sent = 0,
    .num_I_frames_received = 0,
    .num_DISC_sent = 0,
    .num_DISC_received = 0,
    .num_duplicated_frames = 0,
    .num_retransmissions = 0,
    .num_timeouts = 0,
    .num_invalid_BCC1_received = 0,
    .num_invalid_BCC2_received = 0};

// Function to initialize a state machine with given parameters
void create_state_machine(struct state_machine *machine, enum state_machine_type type, unsigned char control_byte, unsigned char address_byte, enum state_machine_state state)
{
    machine->type = type;                          // Set the type of state machine
    machine->control_byte = control_byte;          // Set the control byte
    machine->address_byte = address_byte;          // Set the address byte
    machine->state = state;                        // Initialize state
    memset(machine->buf, 0, sizeof(machine->buf)); // Clear the buffer
    machine->REJ = 0;                              // Initialize REJ flag
    machine->ACK = 0;                              // Initialize ACK flag
    machine->buf_size = 0;                         // Initialize buffer size
    machine->escape_sequence = 0;                  // Initialize escape sequence flag
    machine->duplicate = 0;                        // Initialize duplicate flag
}

// Main function for the state machine processing a byte
void state_machine(struct state_machine *machine, unsigned char byte)
{
    switch (machine->state)
    {
    case START:
        state_machine_START(machine, byte); // Handle START state
        break;

    case FLAG_RCV:
        machine->REJ = 0;                   // Reset REJ flag
        machine->ACK = 0;                   // Reset ACK flag
        machine->BCC2 = 0;                  // Reset BCC2 for new frame
        machine->duplicate = 0;             // Reset duplicate flag
        state_machine_FLAG_RCV(machine, byte); // Handle FLAG_RCV state
        break;

    case A_RCV:
        state_machine_A_RCV(machine, byte); // Handle A_RCV state
        break;

    case C_RCV:
        state_machine_C_RCV(machine, byte); // Handle C_RCV state
        break;

    case BCC1_OK:
        state_machine_BCC1_OK(machine, byte); // Handle BCC1_OK state
        break;

    case STP:
        break; // STP state does nothing
    }
}

// Handle the START state, checking for the FLAG byte
void state_machine_START(struct state_machine *machine, unsigned char byte)
{
    if ((unsigned char)byte == (unsigned char)FLAG)
    {
        machine->state = FLAG_RCV; // Move to FLAG_RCV state
    }
}

// Handle the FLAG_RCV state, checking for the address byte
void state_machine_FLAG_RCV(struct state_machine *machine, unsigned char byte)
{
    if (byte == machine->address_byte)
    {
        machine->state = A_RCV; // Move to A_RCV state
        machine->BCC1 = byte;   // Set BCC1 to address byte
    }
    else if (byte != FLAG)
    {
        machine->state = START; // Invalid byte; reset to START
    }
}

// Handle the A_RCV state, checking for the control byte
void state_machine_A_RCV(struct state_machine *machine, unsigned char byte)
{
    if (byte == machine->control_byte)
    {
        machine->state = C_RCV; // Move to C_RCV state
        machine->BCC1 ^= byte;  // Update BCC1 with control byte
    }
    else if (machine->type == WRITE &&
             ((machine->control_byte == RR0 && byte == REJ1) ||
              (machine->control_byte == RR1 && byte == REJ0)))
    {
        machine->state = C_RCV; // Move to C_RCV for REJ
        machine->REJ = 1;       // REJ received
        machine->BCC1 ^= byte;  // Update BCC1
    }
    else if (machine->type == READ && byte == SET)
    {
        machine->state = C_RCV; // Move to C_RCV for ACK
        machine->ACK = 1;       // SET received
        machine->BCC1 ^= byte;  // Update BCC1
    }
    else if (machine->type == READ &&
             ((machine->control_byte == I_FRAME_0 && byte == I_FRAME_1) ||
              (machine->control_byte == I_FRAME_1 && byte == I_FRAME_0)))
    {
        machine->state = C_RCV; // Move to C_RCV for duplicate
        machine->duplicate = 1; // Duplicate received
        machine->BCC1 ^= byte;  // Update BCC1
    }
    else if (byte == FLAG)
    {
        machine->state = FLAG_RCV; // Return to FLAG_RCV state
    }
    else
```

```c
    }
    else if (byte != FLAG)
    {
        machine->state = START; // Invalid byte; reset to START
    }
}

// Handle the A_RCV state, checking for the control byte
void state_machine_A_RCV(struct state_machine *machine, unsigned char byte)
{
    if (byte == machine->control_byte)
    {
        machine->state = C_RCV; // Move to C_RCV state
        machine->BCC1 ^= byte;  // Update BCC1 with control byte
    }
    else if (machine->type == WRITE &&
            ((machine->control_byte == RR0 && byte == REJ1) ||
             (machine->control_byte == RR1 && byte == REJ0)))
    {
        machine->state = C_RCV; // Move to C_RCV for REJ
        machine->REJ = 1;       // REJ received
        machine->BCC1 ^= byte;  // Update BCC1
    }
    else if (machine->type == READ && byte == SET)
    {
        machine->state = C_RCV; // Move to C_RCV for ACK
        machine->ACK = 1;       // SET received
        machine->BCC1 ^= byte;  // Update BCC1
    }
    else if (machine->type == READ &&
            ((machine->control_byte == I_FRAME_0 && byte == I_FRAME_1) ||
             (machine->control_byte == I_FRAME_1 && byte == I_FRAME_0)))
    {
        machine->state = C_RCV; // Move to C_RCV for duplicate
        machine->duplicate = 1; // Duplicate received
        machine->BCC1 ^= byte;  // Update BCC1
    }
    else if (byte == FLAG)
    {
        machine->state = FLAG_RCV; // Return to FLAG_RCV state
    }
    else
    {
        machine->state = START; // Invalid byte; reset to START
    }
}

// Handle the C_RCV state, checking for the BCC1 byte
void state_machine_C_RCV(struct state_machine *machine, unsigned char byte)
{
    if (byte == machine->BCC1)
    {
        machine->buf_size = 0;    // Reset buffer size
        machine->state = BCC1_OK; // Move to BCC1_OK state
    }
    else if (byte == FLAG)
    {
        machine->state = FLAG_RCV; // Return to FLAG_RCV state
    }
    else
    {
        statistics.num_invalid_BCC1_received++; // Increment invalid BCC1 count
        machine->state = START;                 // Invalid byte; reset to START
    }
}

// Handle the BCC1_OK state, processing incoming data
void state_machine_BCC1_OK(struct state_machine *machine, unsigned char byte)
{
    if (machine->type == READ && !machine->ACK)
    {
        process_read_BCC1_OK(machine, byte); // Process read operation
    }
    else if (byte == FLAG)
    {
        machine->state = STP; // Move to STP on FLAG
    }
    else
    {
        machine->state = START; // Invalid byte; reset to START
    }
}

// Process data in the BCC1_OK state for READ state machine type
void process_read_BCC1_OK(struct state_machine *machine, unsigned char byte)
{
    if (byte == FLAG)
    {
        machine->buf_size--;                            // Remove BCC2 byte from buffer
        machine->BCC2 ^= machine->buf[machine->buf_size]; // Update BCC2

        // Check if the received BCC2 matches the expected BCC2
        if (machine->buf[machine->buf_size] == machine->BCC2)
        {
            machine->state = STP; // Valid frame; move to STP state
        }
        else
        {
            statistics.num_invalid_BCC2_received++; // Increment invalid BCC2 count
            machine->state = STP;                   // Invalid BCC2; move to STP
            machine->REJ = 1;                       // Set REJ to indicate error
        }
    }
    else if (machine->buf_size >= 0 && machine->buf_size < sizeof(machine->buf))
    {
        // Handle byte destuffing
        if (machine->escape_sequence) // If escape sequence was initiated
        {
            machine->escape_sequence = 0; // Reset escape sequence
            byte = (byte == ESC_FLAG) ? FLAG : (byte == ESC_ESC) ? ESC
                                                                 : 0; // Map escaped byte
            if (!byte)
            {
                machine->REJ = 1; // Invalid escape; set REJ
                return;           // Exit processing
            }
        }
        else if (byte == ESC)
        {
            machine->escape_sequence = 1; // Set escape sequence flag
            return;                       // Wait for the next byte
        }

        // Add byte to buffer and update BCC2
        machine->buf[machine->buf_size++] = byte;
        machine->BCC2 ^= byte; // Update BCC2 with the new byte
    }
    else
    {
        machine->state = START; // Buffer overflow; reset to START
    }
}
```

alarm.h

```c
#ifndef _ALARM_H_
#define _ALARM_H_

// Boolean constants for TRUE and FALSE
#define FALSE 0
#define TRUE 1

// Extern declarations for global variables to track alarm state and count
extern int alarm_enabled; // Indicates whether the alarm is currently enabled
extern int alarm_count;   // Counts how many times the alarm has been triggered

// Function declaration for the alarm signal handler
// This function will handle the signal when the alarm goes off
void alarm_handler(int signal);

#endif // _ALARM_H_

```

alarm.c

```c
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include "alarm.h"

// Global variable to track if the alarm is enabled
int alarm_enabled = FALSE;
// Counter for the number of times the alarm has been triggered
int alarm_count = 0;

// Alarm function handler
// This function is called when the alarm signal is received
void alarm_handler(int signal)
{
    alarm_enabled = FALSE; // Disable the alarm
    alarm_count++;         // Increment the alarm trigger count
}
```

serial_port.h

```h
// 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

```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 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

    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);
}

```