Data Link Protocol

RCOM First Project

Bruno Ricardo Soares Pereira de Sousa Oliveira
Rodrigo Albergaria Coelho e Silva

Bachelor's in Informatics and Computing Engineering
Computer Networks

Professor: Hélder Fontes

November 2024

Contents

1 Summary

The aim of this project in Computer Networks was: to develop a Link Layer Protocol to exchange data (ideally a file) from one machine to another through the serial port RS-232. This protocol should be developed in C, use the Stop & Wait protocol, and feature framing, acknowledgment, timeout, retransmission, and byte stuffing mechanisms.

By developing this project and report we were able to develop a reliable communication protocol based on a Stop & Wait mechanism for the Link Layer and Application layer. This allowed us to explore the lower-level implementation of a protocol including basic error detection, as well as test the efficiency of these types of protocols by validating our solution and analyzing the efficiency of the data exchange.

2 Introduction

This report presents the goals, design, and results of the first project of the Computer Networks course. The project focused on creating a Data Link protocol for RS-232 serial communication featuring key concepts like Stop & Wait, framing, error detection, byte stuffing, timeouts, and retransmissions.

The report will be divided into the following sections:

- **Architecture**: describing the functional blocks and interfaces provided by the different sections of our project.
- **Code Structure**: describing the project files in detail, as well as their APIs, data structures and main functions.
- **Main Use Cases**: explaining how the project can be used and the function call sequences associated.
- **Logical Link Protocol**: explaining the functional aspects and implementation of the link layer code.
- **Application Protocol**: explaining the functional aspects and implementation of the application layer code.
- **Validation**: analyzing the validation of the project based on the different tests used.
- **Data Link Protocol Efficiency**: characterizing the protocol's efficiency based on different measurements with varying parameters.
- **Conclusion**: synthesis of the information presented and reflection on the objectives achieved.

Besides these main sections, we will also have two appendix sections, the first one containing the entire source code of the project divided into files, and the second one showcasing graphically the statistical results of the project.

3 Architecture

This project is divided into three components: the **serial port**, the **data link layer**, and the **application layer**, listed in increasing distance from the hardware.

The **serial port** module comprises functions (included in the project template) that allow reading bytes from/writing bytes to the serial port, using system calls and the Linux drivers. The serial port is used in **non-canonical mode**, which makes the program read any number of characters, with no restrictions around new lines, and **read with timeout**, making the read operation return a maximum of one byte and with a timeout of 0.1 seconds.

The **data link layer** provides a robust and stable connection through the communication medium, using the serial port module. It performs many operations for this effect, such as:

- **Connection Establishment and Termination** - The transmitter sends special frames to open/close a connection with the receiver.
- **Information Framing** - Packages the information passed to it with an adequate header and trailer to form a frame.
- **Error Detection** - Verifies errors in both the frame header and frame payload, using Block Check Characters (BCCs).
- **Data transparency** - Performs byte stuffing on every byte received, to ensure that no data characters are perceived as a special character (flag or escape character).
- **Acknowledgment and Retransmission** - For each frame received, the program processes it and responds with a response frame, requesting retransmission if errors were detected, therefore using **Stop-and-Wait**.

Finally, the **application layer** uses the data link layer API to transmit files between two computers. This layer is also responsible for dividing the file into **packets**, for transmission through the link layer, establishment of the start and end of the file transmission, and verification of the files' integrity.

4 Code Structure

In this section, we will describe each one of the project source code files, as well as the most relevant functions.

4.1 Serial Port

The files `serial_port.c` and `serial_port.h` contain the functions responsible for communicating directly with the serial port. These were a part of the project template.

4.2 Alarm

The files `alarm.c` and `alarm.h` includes functions for configuring the Linux alarm signals. These files also define a structure (`AlarmStatus`) for storing the alarm status and count.

4.3 Special Bytes

The file `special_bytes.h` defines the values of the special values used in the protocol, such as the value of the frame flag and control fields.

4.4 State Machines

The files `state.c` and `state.h` implement the state machine functions used on the data link layer on frame reception. The main functions are `nextSOrUFrameState` and `nextIFrameState`, which define two different state machines, one for parsing S/U frames and the other for parsing I frames.

The first state machine is the most simple, only verifying if the bytes are received in order and the BCC matches, rolling back if one of these conditions does not verify. The second state machine, in addition to what the previous one does for the header, is also responsible for other operations, such as determining the current state in the byte destuffing (checking errors if appropriate), checking if the BCC2 matches the received data and verifying the reception of SET and DISC frames, since the protocol must always respond appropriately to those commands.

4.5 Link Layer

The link layer is implemented over the following files: `frame.c`, `frame.h`, `link_layer.c` and `link_layer.h`.

The files `frame.c` and `frame.h` implement auxiliary functions for reading/writing a single frame from the serial port, such as `readIFrame` and `writeSOrUFrame`, using the functions from the serial port and state machine files. In the case of reading a frame, the functions verify the frame integrity and, if an error is detected, the return value reflects the error that occurred (such as the reception of a DISC frame while reading an I frame). A verification that is done at this level is whether the frame data exceeds the maximum payload size, which is used in the link layer to retransmit the packet. This level is also responsible for performing byte stuffing/destuffing.

The files `link_layer.c` and `link_layer.h` define the main functions for the link layer: `llopen` for opening the connection, `llread` for reading a packet, `llwrite` for sending a packet and `llclose` for closing the connection. The files also define the max payload size of an I frame (in the `MAX_PAYLOAD_SIZE` macro) and the structure that stores the connection parameters (`ConnectionParameters`).

4.6 Application Layer

The application layer functionality is dispersed throughout the files `packet.c`, `packet.h`, `application_layer.c` and `application_layer.h`.

The files `packet.c` and `packet.h` define auxiliary functions for the application layer, more specifically functions for reading/writing control/data packets, using the API provided by the data link layer.

The main function of the application layer is defined in the `application_layer.c` and `application_layer.h` files, which receives the connection parameters as arguments and controls the program to transmit the specified file. This function verifies the file integrity

by checking the packet number sequence in the communication and checking the file information at the end. If an error is detected, the function will return the error immediately.

4.7 Debug

The files `debug.c` and `debug.h` contain logging functions for reporting information and errors throughout the program execution. The function `debugLog` prints debug information to the console if the program is compiled with the `-DDEBUG` flag (i.e. using the debug Makefile rules). The function `errorLog` prints the function name and the error message.

4.8 Statistics

The files `statistics.h` and `statistics.c` contain the functions used to document the communication statistics. They define a `Statistics` structure for storing any information relevant to the statistics, which should be initialized using `initStatistics` and updated throughout the program execution. In the end, the statistics are printed to the console using `printStatistics` and stored in files using `storeStatistics`.

5 Main Use Cases

The final project can be used for sending a file (with reasonable size) between two computers through the RS-232 serial port. For this, it is necessary that both computers are connected through this port and run the code simultaneously. The code has two main modes of execution: **transmitter mode** and **receiver mode**, which can be entered by running `make run_tx` and `make run_rx` respectively. The computer that contains the file to send must always be the transmitter.

Execution starts by passing control to the application layer, where both sides try to establish connection by calling the `llopen` function of the link layer.

In case this connection is successful, execution continues in the application layer where the transmitter will open the file to be sent and construct the control and data packets. These are then sent to the link layer to be sent to the receiver with the `llwrite` function, which takes care of framing, byte stuffing and ensuring that the information is sent correctly.

At the same time, the receiver will start listening to the frames coming from the serial port using the `llread` method. The frames will be analyzed to check for errors in the communication and the data is then passed back to the application layer. Here, the data is unpacked and written to the new file on the receiver side.

After sending/receiving all the data, the transmitter and receiver will call `llclose` to close the serial port connection gracefully.

To view more information about the protocol efficiency, the code execution and final results, we can use some alternative execution commands for debug and statistics. These commands are `make run_(tx|rx)_dbg` or `make run_(tx|rx)_stats`.

6 Logical Link Protocol

The link layer protocol is the lowest level we used in this project to the actual physical layer, after the serial port. The implementation of the link layer spans across multiple files: `link_layer` for general functions, `frame` for assembling and disassembling the frames, `state` for controlling the state machine logic, and `alarm` for timeouts on retransmissions. Nonetheless, the main functionalities of this layer are to establish the connection with `llopen`, to send and read the frames `llwrite` and `llread` and, in the end, to close the connection `llclose`.

6.1 Connection Establishment

The connection establishment is done using the `llopen` function, which is intended to be called at the start of the program. The function will fail if a connection is still open.

At this step, the normal control flow will be formed with the transmitter sending a SET frame and the receiver responding with a UA frame. The reception of an acknowledgment by the transmitter is subject to a timeout, sending a frame each time a timeout is exceeded, with a maximum number of retries. No timeout is performed on the receiver side. At this level, if any other frame is received, or if it has an error in the header, it will be ignored.

6.2 Data Transmission

This step is done with the `llwrite` function. It is only meant to be used by the transmitter, failing if used by the receiver or if the connection is closed.

For each call, it will send an I frame with the packet data, properly byte stuffed, and waits for the receiver's response. Each frame is numbered 0 or 1, and for each frame, the function expects the frame RR with the number of the next frame to transmit, finishing successfully in this case. If the function receives an RR frame with the number of the frame it sent (meaning that the receiver received the frame out of order) or a REJ frame (meaning the receiver detected errors in the frame), the I frame will be resent. Similarly, if no response is detected within a timeout period, the frame will be resent, but this time, up to a maximum number of times. If no retransmission limit is reached, the function will only return on the successful transmission of the frame.

6.3 Data Reception

This phase is carried out by the `llread` function, which should only be used by the receiver and will return an error if used by the transmitter or if the connection is closed.

This function will wait for an I frame and process it, performing byte destuffing appropriately. If the frame payload has an error, such as a BCC mismatch, and it has the expected frame number it will respond to the transmitter with a REJ frame and retry. If the frame number is not the expected, the function responds with an RR frame with the expected frame number and retries. Otherwise, an RR frame with the next frame number will be sent and the function will return.

If an I frame is not received, then the function will process the S/U frame. If it is a SET frame (the transmitter opens the connection), it will respond with a UA frame and retry to read an I frame. If it is a SET frame, it will initiate the disconnection sequence and terminate

the function with an error (connection closed unexpectedly). Otherwise, the frame will be ignored.

6.4 Connection Termination

To end the connection, both the transmitter and the receiver should call the `llclose` function. It will fail if there is no open connection.

The control flow of this step is almost as simple as the connection establishment: the transmitter sends a DISC frame, and the receiver responds with a DISC frame, which the transmitter acknowledges with a UA frame. As in the connection establishment, timeouts are applied in the transmitter receptions.

7 Application Protocol

The application layer is responsible for managing the files (sent and received), assembling the control and data packets and passing the control to the link layer. The application layer protocol is divided into two main files, the general `application_layer` broader control and the `packet` for assembling and disassembling the packets.

When the program starts, the application layer will construct the structure with connection parameters and call `llopen` to start the connection.

The transmitter will open the file to be sent, extract some metadata, assemble and send the first control packet using `writeControlPacket`. After this, the `sendFile` function is responsible for repeatedly reading a section of the file and calling `writeDataPacket` to assemble and send the sequence of data packets. When the file is fully sent, the final control packet mus be sent using `writeControlPacket` called once again. Finally, the file is closed and the connection is terminated by calling `llclose`.

The receiver side of the application layer will mirror all the actions. To start, it will create the file and call `readControlPacket` to retrieve the initial metadata. After this, the `receiveFile` function will repeatedly read one section of the file with `readDataPacket` and write it to the new file. When the file is fully sent, the `readControlPacket` is called once again to check the end packet. Finally, the file is closed and the connection is terminated by calling `llclose`.

During all of these steps, some verifications are executed to make sure everything is behaving as expected, for instance, there are no packets lost during transmission, the files are correctly opened and closed, packets have the expected fields, etc.

8 Validation

For this project, we had to develop the protocol keeping in mind many situations/problems that could happen during the transmission and acknowledgment of the data. To validate our final product, we executed the following tests:

- Transmission under **normal conditions**
- Transmission with **interruption of connection** (simulating unplugging the cable and plugging it back)

- Transmission with **bit errors** in the headers and data sections of the frame (changing the bit error ratio (BER) of the cable)
- Transmission **without acknowledgments** or **with different supervision frames** (manual tests by changing the source code)
- **Retransmission of repeated frames** or **wrong packet numbering** (manual tests by changing the source code)
- Transmission with **different baud rates**, **payload sizes** and **propagation times** (and even a combination of these factors to simulate real-world situations)
- Transmission with **different files** (with different file sizes)

For all of these situations the protocol had the expected behavior, ensuring its robustness and functionality. Most of the time, the file was sent successfully without errors, except when the ber was too high, leading to undetectable errors in the transmission. The graph for the efficiency of some of these tests are shown in the Appendix: Efficiency Graphs.

9 Data Link Protocol Efficiency

As a final part of the project, we had to analyze the efficiency of the protocol by comparing the number of good/useful data bits sent by second with the maximum baud rate of the serial port. The ratio between these two is the efficiency of the protocol, which we calculated for different situations like the ones in the previous section.

Each of the following subsections will focus on the statistical characterization and analysis of one these situations according to the concepts we studied about Computer Networks. The graphs for the analysis can be found in the Appendix: Efficiency Graphs.

9.1 Payload Size Variation

One of the test we did was changing the maximum size of the payload sent by the application layer (the size of the sections of the file to be transmitted). This showed that the efficiency stays approximately the same between 74% to 79%. Since there is no bit error ratio, the efficiency depends solely on the overhead of the control sections of the communication protocol.

Despite staying relatively close, we can still see that, for larger payload sizes, the communication will send less control bytes, therefore having less overhead and slightly improving the performance to around 79%.

9.2 Baud Rate Variation

One other test we did was changing the baud rate of the cable to check if it had any impact on the results. As expected, this did not have any major impact on the efficiency of the communication protocol.

By changing the baud rate, we are basically allowing a different number of bits to be sent in the same amount of time. Although increasing the baud rate while keeping the

same frames that are being sent will lead to a faster communication time, the increase is proportional to the baud rate, leading to similar efficiency results.

9.3 Propagation Time Variation

Another test was changing the propagation time of the communication protocol to simulate communication across longer distances. For this, we altered the configurations of the cable and tested for different values and payload sizes to check the efficiency.

In this case, we were able to verify something that was already expected. When the propagation time is very small, any payload size will result in approximately similar efficiency results (as we had verified on the previous graph). However, considering no bit errors, if there is a propagation delay it is more efficient to have fewer exchanges of data. The obvious way to reduce the data exchanges is to increase the payload size of each packet sent.

9.4 Bit Error Ratio Variation

The final situation we analyzed was the impact of bit errors and different payload sizes in the efficiency which allowed us to verify some theories. The efficiency is usually a lot smaller when there are bit errors (compared to no bit errors) because there are retransmissions of packets, leading to more data exchanged for the same original file.

We can also easily verify that for the same bit error ratio, the larger the payload size, the more retransmissions will happen and consequently the smaller the efficiency of the protocol. Therefore, for larger bit error ratios, it is better to have a smaller payload size.

One thing to note is that some errors were noticeable in the file for ratios over 10^{-4}, however, due to the way the protocol is implemented, these errors are deemed undetectable.

We can also point out that for bit error ratios over 10^{-4}, communication with payload size 4096 takes too long and some framing/packet errors may lead to the abortion of communication. A similar situation happened for the ber=10^{-3} and payload size 1024, where we managed to complete one try that took over 30 minutes to complete the communication (with noticeable errors).

10 Conclusion

In conclusion, this project allowed us to deepen our understanding of fundamental link-layer concepts, such as data framing, error detection and correction, and reliable data transfer protocols. Developing a Link Layer Protocol from scratch provided hands-on experience with essential mechanisms like acknowledgment handling, timeouts, and retransmissions, which are critical for ensuring reliable communication.

Building a separate and isolated Application Layer to prepare the data packets and interact with this link layer also taught us more about modularizing and layering code and defining APIs to interact between layers.

Overall, the project successfully met its objectives, giving us practical insight into the complexities of computer networks protocol design and the importance of robust and modular coding strategies for low level projects.

A Appendix: Source Code

A.1 Header files

alarm.h

```c
#ifndef _ALARM_H_
#define _ALARM_H_

typedef struct {
    int enabled;
    int count;
} AlarmStatus;

extern AlarmStatus alarmStatus;

// Configures the alarm and initiates its values.
void configAlarm();

// Restarts the alarm to activate after the specified seconds.
void resetAlarm(unsigned int timeout);

// Stops the alarm.
void stopAlarm();

#endif  // _ALARM_H_
```

application_layer.h

```c
// 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_
```

debug.h

```c
#ifndef _DEBUG_H_
#define _DEBUG_H_

// Logs debug information to the console.
// This function will only output to the console if the program is compiled with the
↪  flag -DDEBUG
void debugLog(const char *format, ...);
```

```c
// Logs errors to the console, using the function name and custom messages.
void errorLog(const char *funcName, const char *messageFormat, ...);

#endif // _DEBUG_H_
```

frame.h

```c
#ifndef _FRAME_H_
#define _FRAME_H_

#include <stdint.h>

#include "alarm.h"

#define SU_FRAME_SIZE     5 // Minimum frame size of S or U frames
#define I_FRAME_BASE_SIZE 5 // Minimum frame size of I frame
#define MAX_BCC2_SIZE     2 // Maximum size of BCC 2

// Reads a S/U frame, checking the address field and extracting the control field.
// Returns 1 on success and a negative value otherwise.
int readSOrUFrame(uint8_t addressField, uint8_t *controlField, int timeout);

// Reads an I frame, checking the address field and extracting the frame number and data
↪  integrity.
// Returns 1 on success and a negative value on error, meaning:
//    -1 ⇒ Unknown error
//    -2 ⇒ Error in data integrity (BCC2 mismatch or data size exceeded)
//    -3 ⇒ Reception of SET frame
//    -4 ⇒ Reception of DISC frame
int readIFrame(uint8_t addressField, uint8_t *frameNumber, uint8_t *data);

// Writes any type of frame into the serial port.
// Returns 1 on success and a negative value otherwise.
int writeFrame(const uint8_t *frame, int frameSize);

// Writes a Supervision or Unnumbered frame.
// Returns 1 on success and a negative value otherwise.
int writeSOrUFrame(uint8_t addressField, uint8_t controlField);

// Writes an Information frame.
// Returns the number of bytes written.
int writeIFrame(uint8_t addressField, uint8_t frameNumber, const uint8_t* data, int
↪  dataSize);

#endif  // _FRAME_H_
```

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 4096

// 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_
```

packet.h

```
#ifndef _PACKET_H_
#define _PACKET_H_

#include <stdint.h>

#include "link_layer.h"

#define BASE_CONTROL_PACKET_SIZE     9                                          //
↪  Control packet base size
#define BASE_DATA_PACKET_SIZE        4                                          //
↪  Data packet base size
#define MAX_FILENAME_SIZE            (MAX_PAYLOAD_SIZE - BASE_CONTROL_PACKET_SIZE)  //
↪  Control packet max filename length
#define MAX_DATA_PACKET_PAYLOAD_SIZE (MAX_PAYLOAD_SIZE - BASE_DATA_PACKET_SIZE)     //
↪  Data packet max payload size

// Reads a control packet from the link layer.
// Returns 1 on success and a negative value otherwise.
int readControlPacket(uint8_t* controlField, uint32_t *filesize, char *filename);

// Reads a data packet from the link layer.
// Returns 1 on success and a negative value otherwise.
```

```c
int readDataPacket(uint8_t *sequenceNumber, uint8_t *packetData);

// Writes a control packet to the link layer.
// Returns 1 on success and a negative value otherwise.
int writeControlPacket(uint8_t controlField, uint32_t filesize, const char *filename);

// Writes a data packet to the link layer.
// Returns 1 on success and a negative value otherwise.
int writeDataPacket(uint8_t sequenceNumber, uint16_t packetDataSize, const uint8_t
↪  *packetData);

#endif  // _PACKET_H_
```

serial_port.h

```c
// 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_
```

special_bytes.h

```c
#ifndef _SPECIAL_BYTES_H_
#define _SPECIAL_BYTES_H_

#define FLAG 0x7E  // Flag byte

#define A1   0x03  // Address field for commands from the transmitter or replies from
↪  the receiver
#define A2   0x01  // Address field for commands from the receiver or replies from the
↪  transmitter

#define SET    0x03          // SET frame control field
#define UA     0x07          // UA frame control field
#define RR(n)  (0xAA + (n))  // RR<n> frame control field
#define REJ(n) (0x54 + (n))  // REJ<n> frame control field
#define DISC   0x0B          // DISC frame control field
```

```
#define C(n) (0x80 * ((n) & 1))  // Information frame n control field

#define ESC       0x7D  // Escape octet
#define ESC2_FLAG 0x5E  // Escaped flag 2nd byte
#define ESC2_ESC  0x5D  // Escaped ESC 2nd byte

#define TLV_FILESIZE_T 0  // Control packet file size type octet
#define TLV_FILENAME_T 1  // Control packet filename type octet

#define CF_START 1  // Start control packet control field
#define CF_DATA  2  // Data packet control field
#define CF_END   3  // End control packet control field

#endif  // _SPECIAL_BYTES_H_
```

state.h

```
#ifndef _STATE_H_
#define _STATE_H_

#include <stdint.h>

typedef enum {
    STATE_START,
    STATE_FLAG_RCV,
    STATE_A_RCV,
    STATE_C_RCV,
    STATE_SET_C_RCV,
    STATE_DISC_C_RCV,
    STATE_BCC1_OK,
    STATE_SET_BCC1_OK,
    STATE_DISC_BCC1_OK,
    STATE_DATA,
    STATE_DATA_ESC,
    STATE_DATA_STUFF,
    STATE_DATA_WRT_STUFF,
    STATE_DATA_ESC_WRT_STUFF,
    STATE_STOP,
    STATE_STOP_SET,
    STATE_STOP_DISC,
    STATE_BCC2_BAD,
    STATE_STUFF_BAD,
} State;

// Returns true if the given state is a data state, and false otherwise.
int isDataState(State state);

// Calculates the next state, in the parsing of a S/U frame.
State nextSOrUFrameState(State state, uint8_t byte, uint8_t addressField, uint8_t
↪  *controlField);

// Calculates the next state, in the parsing of an I frame.
State nextIFrameState(State state, uint8_t byte, uint8_t addressField, uint8_t
↪  *frameNumber, uint8_t xor);

#endif  // _STATE_H_
```

statistics.h

```c
#ifndef _STATISTICS_H_
#define _STATISTICS_H_

#include <time.h>

#include "link_layer.h"

#ifndef STATISTICS
#define STATISTICS 0
#endif

typedef struct {
    LinkLayerRole role;
    unsigned int baudrate;
    struct timespec start;
    struct timespec end;
    unsigned int dataBytes;
    unsigned int totalBytes;
    unsigned int totalFrames;
    unsigned int badFrames;
    unsigned int totalRej;
    unsigned int totalTimeouts;
} Statistics;

extern Statistics statistics;

// Initiates the statistics structure.
void initStatistics(const LinkLayer *connectionParameters);

// Prints the collected statistics to the console.
void printStatistics();

// Stores the statistics into a file.
// Returns 1 on success and a negative value otherwise.
int storeStatistics();

#endif  //_STATISTICS_H_
```

A.2 Source files

alarm.c

```c
#include "alarm.h"

#include <unistd.h>
#include <signal.h>
#include <stdio.h>

AlarmStatus alarmStatus;

// Alarm function handler
void alarmHandler(int signal) {
    alarmStatus.enabled = 0;
    alarmStatus.count++;
}

void configAlarm() {
    alarmStatus.enabled = 0;
    alarmStatus.count = 0;
```

```
    (void)signal(SIGALRM, alarmHandler);
}

void resetAlarm(unsigned int timeout) {
    alarm(timeout);
    alarmStatus.enabled = 1;
}

void stopAlarm() {
    alarm(0);
    alarmStatus.enabled = 0;
}
```

application_layer.c

```
// Application layer protocol implementation

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

#include "application_layer.h"
#include "link_layer.h"
#include "special_bytes.h"
#include "packet.h"
#include "debug.h"
#include "statistics.h"

uint32_t getFileSize(FILE *file) {
    fseek(file, 0, SEEK_END);
    uint32_t size = ftell(file);
    rewind(file);
    return size;
}

int sendFile(FILE* file) {
    uint8_t sequenceNumber = 0;
    uint16_t payloadSize = 0;
    uint8_t buf[MAX_DATA_PACKET_PAYLOAD_SIZE];

    payloadSize = fread(&buf, 1, MAX_DATA_PACKET_PAYLOAD_SIZE, file);
    while (payloadSize > 0) {

        int r = writeDataPacket(sequenceNumber, payloadSize, buf);
        if (r < 0) {
            return -1;
        }
        if (r < payloadSize) {
            errorLog(__func__, "Written data packet smaller than payload size");
            return -1;
        }

        statistics.dataBytes += payloadSize;

        payloadSize = fread(&buf, 1, MAX_DATA_PACKET_PAYLOAD_SIZE, file);
        sequenceNumber = (sequenceNumber + 1) % 100;
    }
    return 1;
}

int receiveFile(FILE* file, uint32_t filesize) {
    uint8_t data[MAX_PAYLOAD_SIZE];
    int dataSize;
    uint8_t sequenceNumber = 0, sequenceNumberCheck = 0;
```

```
    uint32_t totalDataSize = 0;

    while (totalDataSize < filesize) {
        dataSize = readDataPacket(&sequenceNumber, data);
        if (dataSize < 0)
            return -1;

        if (sequenceNumber ≠ sequenceNumberCheck) {
            errorLog(__func__, "Unexpected packet sequence number (received %d, expected
            ↪  %d)", sequenceNumberCheck, sequenceNumber);
            return -1;
        }

        fwrite(data, 1, dataSize, file);
        totalDataSize += dataSize;
        statistics.dataBytes += dataSize;
        sequenceNumberCheck = (sequenceNumberCheck + 1) % 100;
    }
    return 1;
}

void applicationLayer(const char *serialPort, const char *role, int baudRate,
                      int nTries, int timeout, const char *filename) {
    if (serialPort == NULL || role == NULL || filename == NULL) {
        errorLog(__func__, "Null pointer passed as parameter");
        return;
    }

    LinkLayer connectionParameters;
    strncpy(connectionParameters.serialPort, serialPort, 50);
    connectionParameters.role = strcmp(role, "tx") == 0 ? LlTx : LlRx;
    connectionParameters.baudRate = baudRate;
    connectionParameters.nRetransmissions = nTries;
    connectionParameters.timeout = timeout;

    if (llopen(connectionParameters) < 0) {
        return;
    }

    if (connectionParameters.role == LlTx) {

        FILE *file = fopen(filename, "r");
        if (file == NULL) {
            errorLog(__func__, "Couldn't open file to transmit");
            return;
        }

        uint32_t size = getFileSize(file);

        if (writeControlPacket(CF_START, size, filename) < 0)
            return;

        if (sendFile(file) < 0) {
            return;
        }

        if (fclose(file)) {
            errorLog(__func__, "Couldn't close transmitted file");
            return;
        }

        if (writeControlPacket(CF_END, size, filename) < 0)
            return;

    } else {
```

```
        FILE *file = fopen(filename, "w");
        if (file == NULL) {
            errorLog(__func__, "Couldn't open file to receive");
            return;
        }

        uint8_t controlFieldCheck;
        uint32_t filesize1;
        char filename1[MAX_FILENAME_SIZE + 1];
        readControlPacket(&controlFieldCheck, &filesize1, filename1);

        if (controlFieldCheck != CF_START) {
            errorLog(__func__, "Invalid initial packet control field");
            return;
        }

        if (receiveFile(file, filesize1) < 0) {
            return;
        }

        uint32_t filesize2;
        char filename2[MAX_FILENAME_SIZE + 1];
        readControlPacket(&controlFieldCheck, &filesize2, filename2);

        if (controlFieldCheck != CF_END) {
            errorLog(__func__, "Invalid final packet control field");
            return;
        }

        if (filesize1 != filesize2 || strcmp(filename1, filename2) != 0) {
            errorLog(__func__, "Mismatch between start and end packet parameters");
            return;
        }

        uint32_t size = getFileSize(file);
        if (filesize1 != size) {
            errorLog(__func__, "File size mismatch");
            return;
        }

        if (fclose(file)){
            errorLog(__func__, "Couldn't close received file");
            return;
        }
    }

    llclose(STATISTICS);
}
```

debug.c

```
#include "debug.h"

#include <stdarg.h>
#include <stdio.h>

void debugLog(const char *format, ...) {
    #ifdef DEBUG
        va_list args;
        va_start(args, format);
        vprintf(format, args);
        va_end(args);
    #endif
```

```
}

void errorLog(const char *funcName, const char *messageFormat, ...) {
    printf("Error in %s: ", funcName);

    va_list args;
    va_start(args, messageFormat);
    vprintf(messageFormat, args);
    va_end(args);

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

frame.c

```
#include "frame.h"

#include <stdio.h>

#include "state.h"
#include "serial_port.h"
#include "special_bytes.h"
#include "link_layer.h"
#include "debug.h"
#include "statistics.h"

int readSOrUFrame(uint8_t addressField, uint8_t *controlField, int timeout) {
    State state = STATE_START;
    int totalBytes = 0;

    debugLog("Read  <- ");
    while (state != STATE_STOP && (!timeout || alarmStatus.enabled)) {
        uint8_t byte;
        int r = readByteSerialPort(&byte);
        if (r < 0)
            return -1;

        if (r == 1) {
            totalBytes++;
            debugLog(":%02x", byte);

            state = nextSOrUFrameState(state, byte, addressField, controlField);
        }
    }

    if (statistics.role == LlRx) {
        statistics.totalBytes += totalBytes;
        statistics.totalFrames++;
    }

    debugLog(": %d bytes\n", totalBytes);
    return !timeout || alarmStatus.enabled ? 1 : 0;
}

uint8_t decodeByte(uint8_t byte) {
    return byte == ESC2_FLAG ? FLAG : ESC;
}

int readIFrame(uint8_t addressField, uint8_t *frameNumber, uint8_t *data) {
    State state = STATE_START;
    uint8_t xor = 0;
    int dataIndex = -1;
    int totalBytes = 0;
```

```
    debugLog("Read  <- ");

    uint8_t byte;
    uint8_t prevByte;
    while (state ≠ STATE_STOP && state ≠ STATE_BCC2_BAD && state ≠ STATE_STUFF_BAD &&
    ↪  state ≠ STATE_STOP_SET && state ≠ STATE_STOP_DISC) {
        int r = readByteSerialPort(&byte);
        if (r < 0)
            return -1;

        if (r == 1) {
            totalBytes++;
            debugLog(":%02x", byte);
            fflush(stdout);

            state = nextIFrameState(state, byte, addressField, frameNumber, xor);

            if (isDataState(state)) {
                if (dataIndex ⩾ 0) {
                    if (state == STATE_DATA_WRT_STUFF || state ==
                    ↪  STATE_DATA_ESC_WRT_STUFF) {
                        if (dataIndex ⩾ MAX_PAYLOAD_SIZE) {
                            statistics.badFrames++;
                            debugLog("*** Max frame payload size exceeded ***\n");
                            return -2;
                        }
                        data[dataIndex] = decodeByte(prevByte);
                    } else if (state ≠ STATE_DATA_STUFF) {
                        if (dataIndex ⩾ MAX_PAYLOAD_SIZE) {
                            statistics.badFrames++;
                            debugLog("*** Max frame payload size exceeded ***\n");
                            return -2;
                        }
                        data[dataIndex] = prevByte;
                    }
                }

                if (state == STATE_DATA_STUFF) {
                    xor ^= decodeByte(byte);
                } else if (state == STATE_DATA || state == STATE_DATA_WRT_STUFF) {
                    xor ^= byte;
                    dataIndex++;
                } else {  // STATE_DATA_ESC and STATE_DATA_ESC_WRT_STUFF
                    dataIndex++;
                }

                prevByte = byte;

            } else if (state ≠ STATE_STOP) {
                dataIndex = -1;
            }
        }
    }

    statistics.totalBytes += totalBytes;
    statistics.totalFrames++;

    debugLog(": %d bytes\n", totalBytes);

    if (state == STATE_BCC2_BAD) {
        statistics.badFrames++;
        debugLog("*** Bad BCC2 ***\n");
        return -2;  // Data error
    }
    if (state == STATE_STUFF_BAD) {
```

```
        statistics.badFrames++;
        debugLog("*** Bad stuffing detected ***\n");
        return -2;
    }
    if (state == STATE_STOP_SET) {
        statistics.badFrames++;
        debugLog("*** SET frame received ***\n");
        return -3;
    }
    if (state == STATE_STOP_DISC) {
        statistics.badFrames++;
        debugLog("*** DISC frame received ***\n");
        return -4;
    }

    return dataIndex;
}


int writeFrame(const uint8_t *frame, int frameSize) {
    debugLog("Write -> ");

    int bytes;
    for (int i = 0; i < frameSize; i += bytes) {
        bytes = writeBytesSerialPort(frame + i, frameSize);
        if (bytes < 0)
            return -1;

        for (int j = 0; j < bytes; j++)
            debugLog(":%02x", frame[i + j]);
    }

    if (statistics.role == LlTx) {
        statistics.totalBytes += frameSize;
        statistics.totalFrames++;
    }

    debugLog(": %d bytes\n", frameSize);
    return bytes;
}

int writeSOrUFrame(uint8_t addressField, uint8_t controlField) {
    uint8_t frame[SU_FRAME_SIZE] = {FLAG, addressField, controlField, addressField ^
    ↪  controlField, FLAG};

    return writeFrame(frame, SU_FRAME_SIZE);
}

int putByte(uint8_t byte, int *index, uint8_t *frame) {
    if (byte == FLAG) {
        if (*index + 1 ⩾ 2 * MAX_PAYLOAD_SIZE)
            return -1;

        frame[4 + *index] = ESC;
        frame[4 + *index + 1] = ESC2_FLAG;
        *index += 2;

        return 1;
    }

    if (byte == ESC) {
        if (*index + 1 ⩾ 2 * MAX_PAYLOAD_SIZE)
            return -1;

        frame[4 + *index] = ESC;
        frame[4 + *index + 1] = ESC2_ESC;
```

```
        *index += 2;

        return 1;
    }

    frame[4 + *index] = byte;
    (*index)++;

    return 1;
}

int writeIFrame(uint8_t addressField, uint8_t frameNumber, const uint8_t *data, int
↪  dataSize) {
    uint8_t frame[I_FRAME_BASE_SIZE + 2 * (MAX_PAYLOAD_SIZE + 1)];  // Payload and BCC2
    ↪  can be byte stuffed

    frame[0] = FLAG;
    frame[1] = addressField;
    frame[2] = C(frameNumber);
    frame[3] = addressField ^ C(frameNumber);

    uint8_t bcc2 = 0;
    int frameIndex = 0;
    for (int dataIndex = 0; dataIndex < dataSize && frameIndex < 2 * MAX_PAYLOAD_SIZE;
    ↪  dataIndex++) {
        bcc2 ^= data[dataIndex];
        putByte(data[dataIndex], &frameIndex, frame);
    }

    int bccSize = 0;
    putByte(bcc2, &bccSize, frame + frameIndex);
    frame[4 + frameIndex + bccSize] = FLAG;

    if (writeFrame(frame, I_FRAME_BASE_SIZE + frameIndex + bccSize) < 0)
        return -1;
    return frameIndex;
}
```

packet.c

```
#include "packet.h"

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

#include "link_layer.h"
#include "special_bytes.h"
#include "debug.h"

int readControlPacket(uint8_t* controlField, uint32_t *filesize, char *filename) {
    uint8_t packet[MAX_PAYLOAD_SIZE];
    int T1, L1, T2, L2;

    int bytes = llread(packet);
    if (bytes < 0)
        return -1;
    if (bytes < BASE_CONTROL_PACKET_SIZE) {
        errorLog(__func__, "Packet too small to be control packet");
        return -1;
    }

    *controlField = packet[0];
    if (*controlField != CF_START && *controlField != CF_END) {
```

```
        errorLog(__func__, "Control field mismatch (received %d, expected %d or %d)",
        ↪  controlField, CF_START, CF_END);
        return -1;
    }

    // TLV for File Size
    T1 = packet[1];
    if (T1 ≠ TLV_FILESIZE_T) {
        errorLog(__func__, "Unexpected parameter type (received %d, expected %d)", T1,
        ↪  TLV_FILESIZE_T);
        return -1;
    }

    L1 = packet[2];
    if (L1 ≠ 4) {
        errorLog(__func__, "Unexpected parameter length (received %d, expected %d)", L1,
        ↪  4);
        return -1;
    }

    *filesize = (packet[3] << 24) + (packet[4] << 16) + (packet[5] << 8) + packet[6];

    // TLV for File Name
    T2 = packet[7];
    if (T2 ≠ TLV_FILENAME_T) {
        errorLog(__func__, "Unexpected parameter type (received %d, expected %d)", T2,
        ↪  TLV_FILENAME_T);
        return -1;
    }

    L2 = packet[8];
    if (L2 > MAX_FILENAME_SIZE) {
        errorLog(__func__, "Max filename size exceeded");
        return -1;
    }
    memcpy(filename, packet + 9, L2);
    filename[L2] = '\0';

    return 1;
}

uint8_t fromBcd(uint8_t num) {
    return ((num >> 4) * 10) + (num & 0x0F);
}

int readDataPacket(uint8_t *sequenceNumber, uint8_t *packetData) {
    uint8_t packet[MAX_PAYLOAD_SIZE];

    int bytes = llread(packet);
    if (bytes < 0)
        return -1;
    if (bytes < BASE_CONTROL_PACKET_SIZE) {
        errorLog(__func__, "Packet too small to be data packet");
        return -1;
    }

    uint8_t controlField = packet[0];
    if (controlField ≠ CF_DATA) {
        errorLog(__func__, "Control field mismatch (received %d, expected %d)",
        ↪  controlField, CF_DATA);
        return -1;
    }

    *sequenceNumber = fromBcd(packet[1]);

    uint16_t packetDataSize = (packet[2] << 8) + packet[3];
```

```c
    if (packetDataSize > MAX_DATA_PACKET_PAYLOAD_SIZE) {
        errorLog(__func__, "Max data packet size exceeded");
        return -1;
    }

    memcpy(packetData, packet + 4, packetDataSize);

    return packetDataSize;
}

int writeControlPacket(uint8_t controlField, uint32_t filesize, const char *filename) {
    int filenameSize = strlen(filename);
    if (filenameSize > MAX_FILENAME_SIZE) {
        errorLog(__func__, "Max filename size exceeded");
        return -1;
    }

    uint8_t buf[BASE_CONTROL_PACKET_SIZE + MAX_FILENAME_SIZE];

    buf[0] = controlField;

    // TLV for file size
    buf[1] = TLV_FILESIZE_T;
    buf[2] = 4;
    buf[3] = (filesize >> 24) & 0xFF;
    buf[4] = (filesize >> 16) & 0xFF;
    buf[5] = (filesize >> 8) & 0xFF;
    buf[6] = (filesize) & 0xFF;

    // TLV for file name
    buf[7] = TLV_FILENAME_T;
    buf[8] = filenameSize;
    memcpy(buf + 9, filename, filenameSize);

    return llwrite(buf, BASE_CONTROL_PACKET_SIZE + filenameSize);
}

uint8_t toBcd(uint8_t num) {
    return ((num / 10) << 4) + (num % 10);
}

int writeDataPacket(uint8_t sequenceNumber, uint16_t packetDataSize, const uint8_t
↪   *packetData) {
    if (packetDataSize > MAX_DATA_PACKET_PAYLOAD_SIZE) {
        errorLog(__func__, "Max packet data size exceeded");
        return -1;
    }

    uint8_t buf[BASE_DATA_PACKET_SIZE + MAX_DATA_PACKET_PAYLOAD_SIZE];

    buf[0] = CF_DATA;
    buf[1] = toBcd(sequenceNumber);
    buf[2] = (packetDataSize >> 8) & 0xFF;
    buf[3] = (packetDataSize) & 0xFF;
    memcpy(buf + 4, packetData, packetDataSize);

    return llwrite(buf, BASE_DATA_PACKET_SIZE + packetDataSize);
}
```

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] = 1; // Read with timeout
    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.
```

```c
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);
}
```

state.c

```c
#include "special_bytes.h"
#include "state.h"

int isDataState(State state) {
    return state == STATE_DATA || state == STATE_DATA_ESC || state == STATE_DATA_STUFF
        || state == STATE_DATA_WRT_STUFF || state == STATE_DATA_ESC_WRT_STUFF;
}

int isValidSOrUFrameControl(uint8_t byte) {
    return byte == SET || byte == UA || byte == RR(0) || byte == RR(1) || byte == REJ(0)
    ↪  || byte == REJ(1) || byte == DISC;
}

State nextSOrUFrameState(State state, uint8_t byte, uint8_t addressField, uint8_t
↪  *controlField)
{
    switch (state) {
      case STATE_START:
        if (byte == FLAG)
            state = STATE_FLAG_RCV;
        break;

      case STATE_FLAG_RCV:
        if (byte == addressField)
            state = STATE_A_RCV;
        else if (byte ≠ FLAG)
            state = STATE_START;
        break;

      case STATE_A_RCV:
        if (isValidSOrUFrameControl(byte)) {
            *controlField = byte;
            state = STATE_C_RCV;
        } else if (byte == FLAG) {
            state = STATE_FLAG_RCV;
        } else {
            state = STATE_START;
        }
        break;

      case STATE_C_RCV:
        if (byte == (addressField ^ *controlField))
            state = STATE_BCC1_OK;
        else if (byte == FLAG)
            state = STATE_FLAG_RCV;
        else
            state = STATE_START;
        break;
```

```
      case STATE_BCC1_OK:
        if (byte == FLAG)
            state = STATE_STOP;
        else
            state = STATE_START;
        break;

      default:
        break;
    }

    return state;
}

State nextIFrameState(State state, uint8_t byte, uint8_t addressField, uint8_t
↪   *frameNumber, uint8_t xor) {
    switch (state) {
      case STATE_START:
        if (byte == FLAG)
            state = STATE_FLAG_RCV;
        break;

      case STATE_FLAG_RCV:
        if (byte == addressField)
            state = STATE_A_RCV;
        else if (byte ≠ FLAG)
            state = STATE_START;
        break;

      case STATE_A_RCV:
        if (byte == C(0)) {
            *frameNumber = 0;
            state = STATE_C_RCV;
        } else if (byte == C(1)) {
            *frameNumber = 1;
            state = STATE_C_RCV;
        } else if (byte == SET) {
            state = STATE_SET_C_RCV;
        } else if (byte == DISC) {
            state = STATE_DISC_C_RCV;
        } else if (byte == FLAG) {
            state = STATE_FLAG_RCV;
        } else {
            state = STATE_START;
        }
        break;

      case STATE_C_RCV:
        if (byte == (addressField ^ C(*frameNumber)))
            state = STATE_BCC1_OK;
        else if (byte == FLAG)
            state = STATE_FLAG_RCV;
        else
            state = STATE_START;
        break;

      case STATE_SET_C_RCV:
        if (byte == (addressField ^ SET))
            state = STATE_SET_BCC1_OK;
        else if (byte == FLAG)
            state = STATE_FLAG_RCV;
        else
            state = STATE_START;
        break;
```

```
      case STATE_DISC_C_RCV:
        if (byte == (addressField ^ DISC))
            state = STATE_DISC_BCC1_OK;
        else if (byte == FLAG)
            state = STATE_FLAG_RCV;
        else
            state = STATE_START;
        break;

      case STATE_BCC1_OK:
        state = STATE_DATA;
        break;

      case STATE_SET_BCC1_OK:
        if (byte == FLAG)
            state = STATE_STOP_SET;
        else
            state = STATE_START;
        break;

      case STATE_DISC_BCC1_OK:
        if (byte == DISC)
            state = STATE_STOP_DISC;
        else
            state = STATE_START;
        break;

      case STATE_DATA:
      case STATE_DATA_WRT_STUFF:
        if (byte == FLAG)
            state = xor == 0 ? STATE_STOP : STATE_BCC2_BAD;
        else if (byte == ESC)
            state = STATE_DATA_ESC;
        else {
            state = STATE_DATA;
        }
        break;

      case STATE_DATA_ESC:
      case STATE_DATA_ESC_WRT_STUFF:
        if (byte == ESC2_FLAG) {
            state = STATE_DATA_STUFF;
        } else if (byte == ESC2_ESC) {
            state = STATE_DATA_STUFF;
        } else {
            state = STATE_STUFF_BAD;
        }
        break;

      case STATE_DATA_STUFF:
        if (byte == ESC) {
            state = STATE_DATA_ESC_WRT_STUFF;
        } else if (byte == FLAG) {
            state = xor == 0 ? STATE_STOP : STATE_BCC2_BAD;
        } else {
            state = STATE_DATA_WRT_STUFF;
        }
        break;

      default:
        break;
    }

    return state;
}
```

statistics.c

```c
#include "statistics.h"

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

#include "debug.h"

Statistics statistics;

void initStatistics(const LinkLayer *connectionParameters) {
    statistics.role = connectionParameters->role;
    statistics.baudrate = connectionParameters->baudRate;
    statistics.dataBytes = 0;
    statistics.totalBytes = 0;
    statistics.totalFrames = 0;
    statistics.badFrames = 0;
    statistics.totalRej = 0;
    statistics.totalTimeouts = 0;
}

void printStatistics() {
    double totalTime = (statistics.end.tv_sec - statistics.start.tv_sec) * 1e9 +
    ↪  (statistics.end.tv_nsec - statistics.start.tv_nsec);
    double measuredBaudrate = statistics.dataBytes * 8 * 1e9 / totalTime;
    double efficiency = measuredBaudrate / statistics.baudrate;

    printf("\n");
    printf("************ STATISTICS ************\n");
    printf("\n");
    printf("Communication time: %f s\n", totalTime / 1e9);
    printf("Serial port baudrate: %d\n", statistics.baudrate);
    printf("Payload size: %d\n", MAX_PAYLOAD_SIZE);
    printf("Measured baudrate: %f\n", measuredBaudrate);
    printf("Efficiency: %f\n", efficiency);
    printf("\n");

    switch (statistics.role) {
      case LlTx:
        printf("Total bytes transmitted: %d\n", statistics.totalBytes);
        printf("Data bytes transmitted: %d\n", statistics.dataBytes);
        printf("Total frames transmitted: %d\n", statistics.totalFrames);
        printf("Total negative acknowledgements: %d\n", statistics.totalRej);
        printf("Total timeouts: %d\n", statistics.totalTimeouts);
        printf("Frame error ratio: %f\n", (double)(statistics.totalRej +
        ↪  statistics.totalTimeouts) / statistics.totalFrames);
        break;

      case LlRx:
        printf("Total bytes received: %d\n", statistics.totalBytes);
        printf("Data bytes received: %d\n", statistics.dataBytes);
        printf("Total frames received: %d\n", statistics.totalFrames);
        printf("Good frames detected: %d\n", statistics.totalFrames -
        ↪  statistics.badFrames);
        printf("Bad frames detected: %d\n", statistics.badFrames);
        printf("Frame error ratio: %f\n", (double)statistics.badFrames /
        ↪  statistics.totalFrames);
        break;
    }

    printf("\n");
    printf("************************************\n");
}
```

```c
int storeStatistics() {
    FILE *file;
    double totalTime = (statistics.end.tv_sec - statistics.start.tv_sec) * 1e9 +
    ↪  (statistics.end.tv_nsec - statistics.start.tv_nsec);
    double measuredBaudrate = statistics.dataBytes * 8 * 1e9 / totalTime;
    double efficiency = measuredBaudrate / statistics.baudrate;

    if (statistics.role == LlTx) {
        const char *filename = "stats-tx.csv";
        if ((file = fopen(filename, "r")) != NULL) {
            fclose(file);

            file = fopen(filename, "a");
            if (file == NULL) {
                errorLog(__func__, "Couldn't open statistics spreadsheet");
                return -1;
            }

        } else {
            file = fopen(filename, "w");
            if (file == NULL) {
                errorLog(__func__, "Couldn't create statistics spreadsheet");
                return -1;
            }

            fprintf(file, "Bit error ratio,Propagation time,Baudrate,Payload
            ↪  size,Communication time (s),Measured baudrate,Efficiency,Total
            ↪  bytes,Data bytes,Total frames,NACK's,Timeouts,Frame error ratio,Errors
            ↪  in file\n");
        }

        fprintf(file, ",,%d,%d,%f,%f,%f,%d,%d,%d,%d,%d,%f,\n",
            statistics.baudrate,
            MAX_PAYLOAD_SIZE,
            totalTime / 1e9,
            measuredBaudrate,
            efficiency,
            statistics.totalBytes,
            statistics.dataBytes,
            statistics.totalFrames,
            statistics.totalRej,
            statistics.totalTimeouts,
            (double)(statistics.totalRej + statistics.totalTimeouts) /
            ↪  statistics.totalFrames
        );

    } else {
        const char *filename = "stats-rx.csv";
        if (access(filename, F_OK) == 0) {
            file = fopen(filename, "a");
            if (file == NULL) {
                errorLog(__func__, "Couldn't open statistics spreadsheet");
                return -1;
            }

        } else {
            file = fopen(filename, "w");
            if (file == NULL) {
                errorLog(__func__, "Couldn't create statistics spreadsheet");
                return -1;
            }

            fprintf(file, "Bit error ratio,Propagation time,Baudrate,Payload
            ↪  size,Communication time (s),Measured baudrate,Efficiency,Total
            ↪  bytes,Data bytes,Total frames,Good frames,Bad frames,Frame error
            ↪  ratio,Errors in file\n");
```

```
        }

        fprintf(file, ",,%d,%d,%f,%f,%f,%d,%d,%d,%d,%d,%f,\n",
            statistics.baudrate,
            MAX_PAYLOAD_SIZE,
            totalTime / 1e9,
            measuredBaudrate,
            efficiency,
            statistics.totalBytes,
            statistics.dataBytes,
            statistics.totalFrames,
            statistics.totalFrames - statistics.badFrames,
            statistics.badFrames,
            (double)statistics.badFrames / statistics.totalFrames
        );
    }

    return 1;
}
```

B Appendix: Efficiency Graphs

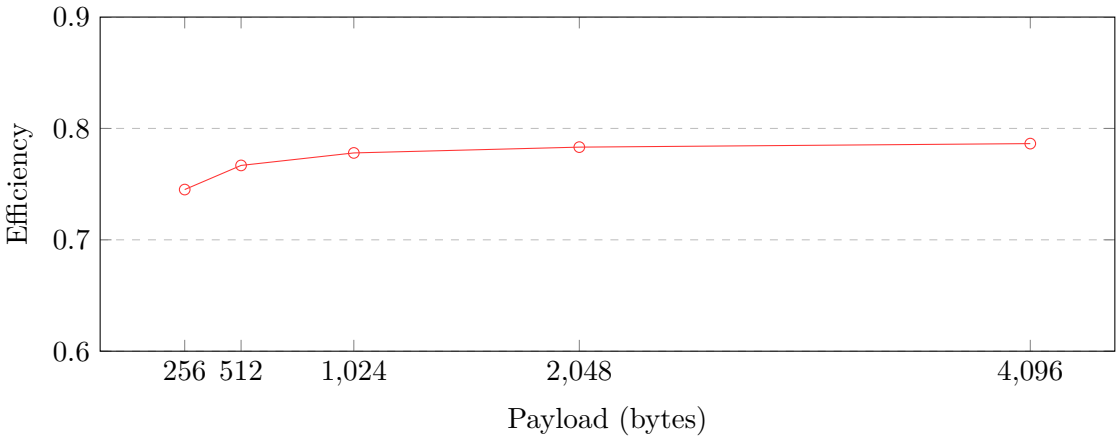

Figure 1: Efficiency analysis of different payload sizes

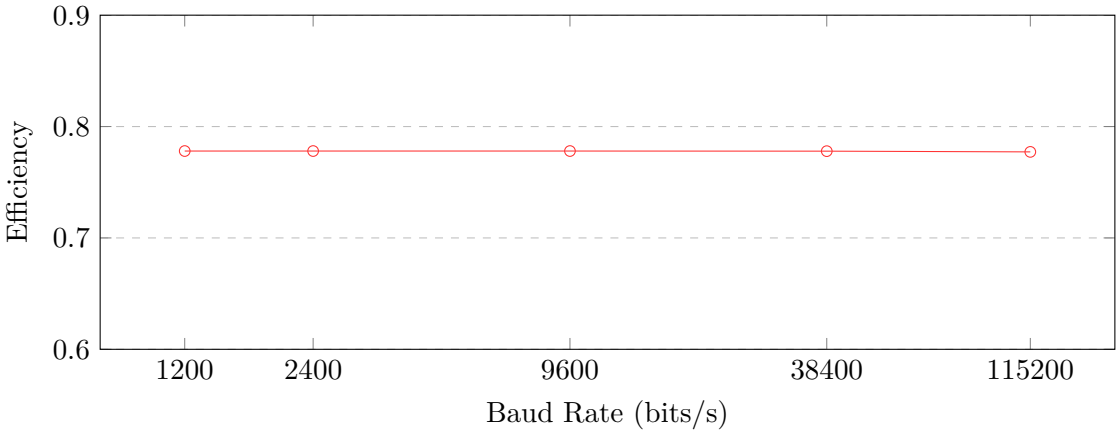

Figure 2: Efficiency analysis of different baud rates

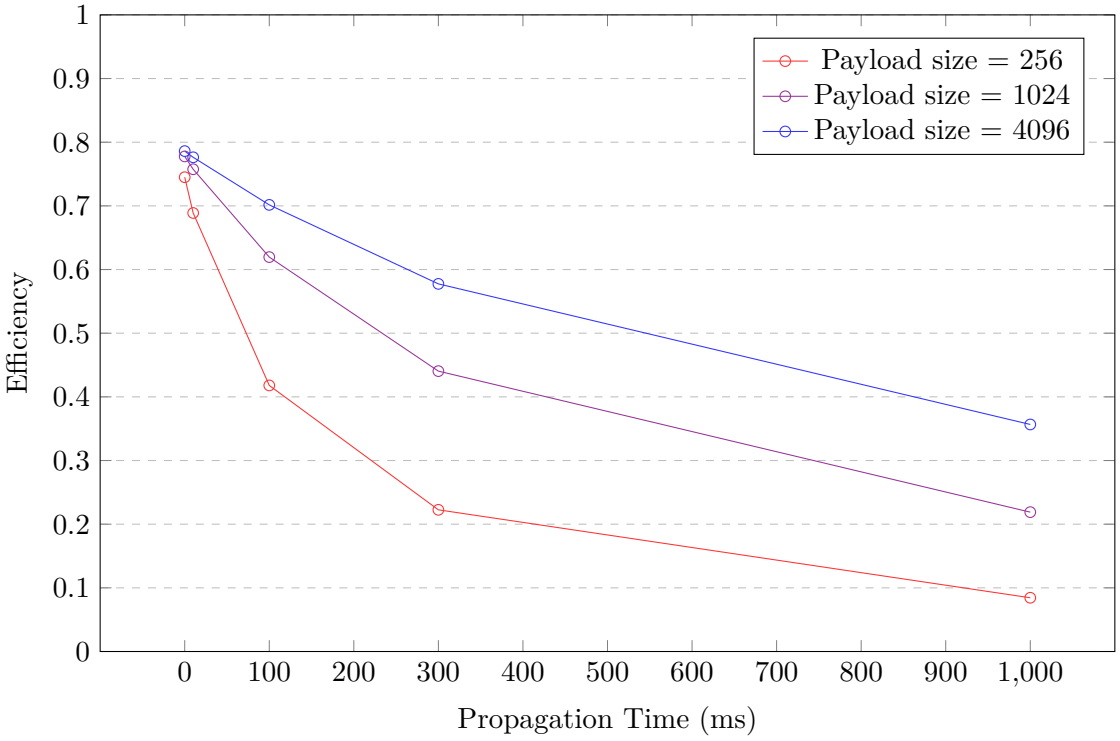

Figure 3: Efficiency analysis of different propagation times and payload sizes

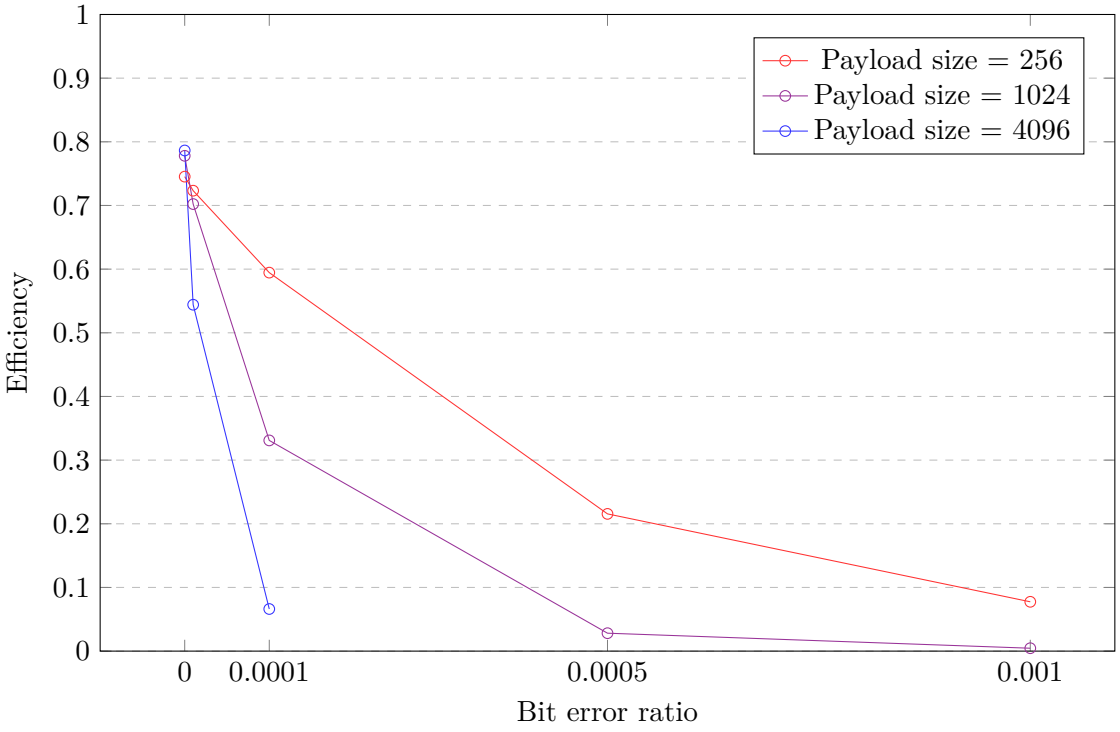

Figure 4: Efficiency analysis of different bit error ratios and payload sizes