Data Link Protocol

Lab 1

Computer Networks 2024/2025

Authors:
António Santos (up202205469@up.pt) - T11
Vanessa Queirós (up202207919@up.pt) - T11

Table of Contents

Summary

This project was created as part of the *Computer Networks* course with the goal of developing a data link protocol for transmitting a file via the RS-232 serial cable.

With this project, we were able to consolidate our knowledge on data transfer and several techniques needed to ensure a reliable communication between the transmitter and receiver.

Introduction

The main goal of this project is to implement a data link protocol to transfer a file stored on one computer to another computer through a RS-232 serial cable. We developed a transmitter and receiver data transfer application to test the protocol.

Throughout the report, there are eight sections, ranging from the code architecture and structure to the overall conclusions of the project:

- **Architecture**: Functional blocks and interfaces.
- **Code Structure**: Presentation of the APIs, main structures and main functions used.
- **Main Use Cases**: Identification of how the project works and function call sequences.
- **Logical Link Protocol**: How the logical link protocol works and identification of the main functional aspects.
- **Application Protocol**: How the application protocol works and identification of the main functional aspects.
- **Validation**: Tests performed and results.
- **Data Link Protocol Efficiency**: Statistical characterization of the protocol's efficiency.
- **Conclusions**: Synthesis and Reflection on the information presented in the previous sections.

Architecture

Functional Blocks

The project has three main layers: the *Application Layer*, the *Link Layer* and the *Serial Port Layer*. Each layer has specific responsibilities, and they work together to ensure the proper transmission of data over the communication channel.

The *Application Layer* is responsible for transferring and receiving control and data packets. This is where the actual data to be sent is divided into packets and passed on to the lower layer, which is the *Link Layer*.

The *Link Layer* is responsible for sending / receiving packets. These packets originally come from the *Application Layer* in the Transmitter side (Tx) and are sent to the Receiver side (Rx). The *Serial Port Layer* was not actually implemented by us, but it is the lowest layer that directly deals with the serial port.

Interfaces

Both the transmitter and receiver have similar interfaces, with slight differences in the statistics:

```
antonioabilio@fedora:~/Desktop/RCOM-Lab/Proj$ ./bin/main /dev/ttyS11 9600 tx penguin.gif
Starting link-layer protocol application
  - Serial port: /dev/ttyS11
  - Role: tx
  - Baudrate: 9600
  - Number of tries: 3
  - Timeout: 4
  - Filename: penguin.gif
Statistics:
Number of dropped packets (TX): 0
Total number of frames that were retransmitted: 0
Total number of timeouts: 0
Number of frames that were sent/received and are valid: 14
Number of frames that were sent/received and are invalid: 0
Total number of frames that were sent/received: 14
```

Figure 1: TX-side interface

```
antonioabilio@fedora:~/Desktop/RCOM-Lab/Proj$ ./bin/main /dev/ttyS10 9600 rx penguin-received.gif
Starting link-layer protocol application
  - Serial port: /dev/ttyS10
  - Role: rx
  - Baudrate: 9600
  - Number of tries: 3
  - Timeout: 4
  - Filename: penguin-received.gif
Tx is reading a file with name: penguin.gif
Statistics:
Number of dropped packets (RX): 0
Number of frames received that were duplicate: 0
Number of frames that were sent/received and are valid: 14
Number of frames that were sent/received and are invalid: 0
Total number of frames that were sent/received: 14
```

Figure 2: RX-side interface

Code Structure

The code structure of the system is organized across two main layers: the *Application Layer* and the *Link Layer*. Each layer exposes a set of functions that work together to facilitate data transmission and reception, ensuring a modular and structured approach to handling communication tasks. The *Application Layer* is responsible for reading (Tx side) data from the file that we want to send and for writing (Rx side) the data received from the *Link Layer* to a file. The main functions of this layer are the following:

```
// This function is responsible for creating the Control Packet.
int createControlPacket()

// This function creates Data Packets from the file data at the
transmitter side.
int createDataPacket()
```

```
// Function that reads the Control Packet.
int readControlPacket()

// Function that reads the Data Packet and writes to the file.
int readDataPacket()

// Function that manages the Rx and Tx side of the application.
int applicationLayer()
```

The *Link Layer* provides an API for the *Application Layer*. These are the main functions of the *Link Layer*:

```
// Function that helps start the connection between Tx and Rx.
int llopen()

// Function that Tx uses to write frames to the serial port.
int llwrite()

// Function that Rx uses to read frames from the serial port.
int llread()

// Function that terminates the connection between Tx and Rx.
int llclose()
```

The *Application Layer* relies on the *Link Layer* to handle the low-level communication tasks. For example, when the Application Layer generates a data packet, it calls `llwrite()` to transmit the packet through the serial port. Similarly, when receiving data, the Application Layer calls `llread()` to retrieve packets, which are then processed or written to a file. The *Link Layer* abstracts the complexities of the serial communication, providing a simple API to the Application Layer for reading, writing, and managing the connection.

Main Use Cases

The function call sequences are different, depending on whether the program is executed as a transmitter or a receiver:

Transmitter

1. *applicationLayer()*
2. *txApplication()*
3. *llopen()*
4. *createControlPacket()*
5. *llwrite()*
6. *createDataPacket()*
7. *llwrite() // This function is called multiple times to write all the data packets*
8. *createControlPacket()*
9. *llwrite()*
10. *llclose()*

Receiver

1. *applicationLayer()*
2. *rxApplication()*
3. *llopen()*
4. *readControlPacket()*
5. *llread()*
6. *readDataPacket()*
7. *llread() // This function is called multiple times. It exits when a Control Packet is read.*
8. *readControlPacket()*
9. *llread()*
10. *llclose()*

The main use case is to send and receive data across two devices using the serial port.

Logical Link Protocol

The Link Layer is the layer that sits between the *Application Layer* and the *Serial Port Layer*. It is also the protocol that the *Application Layer* can rely on to send and receive data by using its API.

Other than the expected functionality of this protocol (establishing connection, writing data, reading data, closing the connection) the protocol does more things. Two of the most important are: the validation of the data that is being received through the use of BCCs and the Stop and Wait Method that allows the protocol to be resilient against malformed packets (BCCs not being the same, missing data), packet drops and the possibility of receiving duplicate data.

After connection establishment, first the transmitter uses *llwrite()*, which takes the packet from the layer above and adds a header that contains a *Flag*, an *Address Field*, a *Control Byte* and a *BCC1* (guarantees that the header has no errors), calculates a *BCC2* for the packet that is going to be sent, does the *Byte Stuffing* part and finally adds another *Flag* to the end. After this is done the packet is sent through the serial port. Now the transmitter waits for a positive or negative acknowledgement (*Ack* or a *Nack*) that is sent by the receiver. If a positive acknowledgment is received then we return to the *Application Layer* and prepare to send another packet. However, if we receive a negative acknowledgement we will retransmit the packet and wait for a response once again. Finally, if in a certain amount of time we do not receive any type of acknowledgment, then we try to retransmit the packet and if a certain amount of tries is reached we return to the *Application Layer* which in turn stops the program execution.

On the receiver side, after connection establishment, when the *Application Layer* calls *llread()*, the program starts to read the bytes from the serial port and validates them. First it validates the header using *BCC1* and right away it decides if the packet is going to be read until the end or if we should stop early and send a negative acknowledgement. If however, the *BCC1* is valid, it reads until we reach a *Flag* where we go to the data validation phase. In this phase the *Byte Destuffing* and *BCC2* validation take place, and the program determines whether the received frame is valid or not. If the validation is successful, an *Ack* is sent to confirm proper reception of the data. Otherwise, a *Nack* is sent to request retransmission due to errors in the received packet.

By carefully managing these aspects, the *Link Layer* plays a crucial role in ensuring that the data exchange between the *Application Laye*r and the *Serial Port* is both reliable and efficient. However, as discovered while testing, this still may not be enough and some packets may have errors that even using the mechanisms described above are not detected.

Application Protocol

The application layer is the highest-level layer and is responsible for sending the contents of the file as data packets that the receiver will then reassemble into a file. While this layer doesn't directly handle transmission errors, it relies on the *Link Layer* (one layer below the *Application Layer*) as an API to facilitate data transfer.

Initially, both the transmitter and the receiver call the *Link Layer* function *llopen()* to establish a connection. Then the transmitter sends a starting control packet, which is a packet that stores information about the data to be sent. This is done in TLV (type, length value) format. After sending the starting control packet, the transmitter sends the data packets. The file is divided into several data packets of a size that is specified in the *Application Layer*. Based on connection stability and noise levels, a larger or smaller packet size may be more appropriate to optimize data transfer efficiency. Once the file transfer is complete, an additional control packet is sent. This control packet is identical to the first one, except that the first byte is modified to indicate that it is an ending control packet. All these packets are read by the receiver through the functions *readControlPacket()* and *readDataPacket()*, depending on whether the received packet is a control packet or a data packet.

After the transfer is complete, both the transmitter and the receiver call the *Link Layer* function *llclose()* to close the connection and terminate the program. The implementation of these functions can be viewed in **Appendix IV - link_layer.c** and **Appendix II - application_layer.c**.

Validation

In order to ensure that our program functions correctly and meets all requirements, we performed the following tests in class:

- Serial port interruption, leading to timeouts
- Introduction of noise in the serial port, leading to an increased bit error rate (and frame error rate)

Outside of class, we performed some additional tests:

- Sending files with different sizes
- Sending files using different baud rates
- Sending files using different packet sizes

We managed to pass all the tests with the expected results. However, because the BCC error checking method does not guarantee a correctness of 100% (There is always a possibility of collisions for the BCC2. This probability increases as the number of bytes that are sent in one packet increases.), there were cases where one or two frames were not completely correct, leading to a slightly distorted image. The higher the BER, the greater the probability of this happening. For example, testing with 0.0008 BER led us to having a distorted penguin due to some bytes being altered in the file. The data packets weren't rejected because the combination of these numbers lead to the XOR (BCC2) being the same as the original.

View it here -> https://github.com/vanessa-sbq/RCOM-Lab/blob/main/README.md

Data Link Protocol Efficiency

The Stop-and-Wait ARQ protocol was used to manage data integrity and ensure that the received data is correct. Upon receiving a frame, the receiver checks its validity. If the data is invalid, it sends a *NACK* to the transmitter, indicating that the frame needs to be retransmitted. If the frame is valid, the receiver sends an *ACK* to confirm that the reception was successful. We tested our program's efficiency for three parameters: FER (frame error ratio), baudrate and frame size.

FER Variation

Varying the FER essentially means that the number of *NACK*s sent by the receiver will be higher, leading to a higher number of retransmitted frames, since a frame with errors should not be accepted by the receiver. We tested the program with different FERs and noticed a significant difference in the program's efficiency. For a baudrate of 38400, we get the values in **Appendix VIII - FER Variation.** We concluded that the larger the FER, the lower the efficiency.

Baudrate Variation

Increasing the baudrate means that we can send more information through the serial port at once. However, this does not mean that using a higher baudrate is more efficient.
In our case (frame size of 1000 bytes), the efficiency peaked at baudrates between 1800 and 2400 bits/s. For a FER (Frame Error Rate) of 0, we get the values in **Appendix IX - Baudrate Variation.** We can also conclude that the baudrate and the total transmission time of the file are inversely proportional.

Frame Size Variation

By varying the frame size, we noticed that the efficiency increases with higher frame sizes. However, this might not be the best solution for higher FERs, since a bigger frame is more likely to contain frame errors rather than a smaller frame, leading to a higher number of retransmissions.
For a baudrate of 38400, we get the values in **Appendix X - Frame Size Variation.**

Conclusions

In conclusion, this project provided valuable insights into the protocols involved to enable reliable data transmission. Throughout this report, we presented and demonstrated the development and testing of a file transfer program that handles packets, manages connection stability and validates data integrity, through different mechanisms, such as *Stop-and-Wait* and *Byte Stuffing*.

Overall, we enhanced our knowledge and understanding on the different network layers involved in order to make transmission reliable and efficient.

Appendices

Appendix I - *application_layer.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_
```

Appendix II - *application_layer.c*

```
// Application layer protocol implementation
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include "application_layer.h"
#include "link_layer.h"

// Definitions for Control Packets
#define CtrlPacketStart 1
#define CtrlPacketEnd 3

#define CSTART 1
#define CDATA 2
#define CEND 3

// Definitions for Data Packets
#define partitionSize 996
unsigned char sequenceNumber = 0;  // Between 0 and 99


/**
```

```
* Creates a control packet (start or end)
* controlPacket - array to which the packet is written to
* currentSize - current size of the control packet (while it is being
written to)
* cpt - should have values CSTART or CEND (start or end control packet)
* fileSize - size of the file to be sent
* fileName - name of the file to be sent
* returns a controlPacket on success
*         NULL on error
*/
unsigned char* createControlPacket(unsigned char* controlPacket, int*
currentSize, int cpt, long fileSize, const unsigned char* fileName) {
   if (controlPacket == NULL) return NULL;

   if (cpt == CSTART) controlPacket[0] = CSTART; // Dealing with a Start
Control Packet.
   else {
       controlPacket[0] = CEND; // Dealing with a End Control Packet.
       return controlPacket; // The ending packet is the same as the
starting packet. Only difference is the first value.
   }

   // TLV coded long
   char byteData[4];
   byteData[0] = (fileSize >> 24) & 0xFF; // Most significant byte
   byteData[1] = (fileSize >> 16) & 0xFF;
   byteData[2] = (fileSize >> 8) & 0xFF;
   byteData[3] = fileSize & 0xFF; // Least significant byte

   // Type
   (*currentSize)++;
   controlPacket = (unsigned char*)realloc(controlPacket, (*currentSize)
* sizeof(unsigned char));
   controlPacket[(*currentSize) - 1] = 0; // Filesize

   // Length
   (*currentSize)++;
   controlPacket = (unsigned char*)realloc(controlPacket, (*currentSize)
* sizeof(unsigned char));
   controlPacket[(*currentSize) - 1] = 4;

   for (int i = 0; i < 4; i++) {
       // Value
       (*currentSize)++;
       controlPacket = (unsigned char*)realloc(controlPacket,
(*currentSize) * sizeof(unsigned char));
       controlPacket[(*currentSize) - 1] = byteData[i];
   }

   // TLV coded filename
   int fileNameSize = (int)strlen((const char*) fileName);

   // Type
   (*currentSize)++;
   controlPacket = (unsigned char*)realloc(controlPacket, (*currentSize)
* sizeof(unsigned char));
```

```
   controlPacket[(*currentSize) - 1] = 1; // Filename

   // Length
   (*currentSize)++;
   controlPacket = (unsigned char*)realloc(controlPacket, (*currentSize)
* sizeof(unsigned char));
   controlPacket[(*currentSize) - 1] = fileNameSize;

   //Value
   for (int i = 0; i < fileNameSize; i++) {
       (*currentSize)++;
       controlPacket = (unsigned char*)realloc(controlPacket,
(*currentSize) * sizeof(unsigned char));
       controlPacket[(*currentSize) - 1] = fileName[i];
   }
   return controlPacket;
}


/**
* Creates a data packet according to the specification
* dataPacket[] - data packet array to be written
* currentSize - size of the data packet to be written
* fd - file descriptor
* returns 1 on success
*         0 if no bytes are read (nothing left to read)
*        -1 on error
*/
int createDataPacket(unsigned char* dataPacket[], int* currentSize, int*
fd) {
   unsigned int accumulatorOfBytesRead = 0;
   unsigned char byte = 0x00;

   (*dataPacket)[0] = CDATA; // Control Data

   // Sequence Number
   (*currentSize)++;
   (*dataPacket) = (unsigned char*)realloc((*dataPacket), (*currentSize)
* sizeof(unsigned char));
   (*dataPacket)[(*currentSize) - 1] = sequenceNumber;

   // L2
   (*currentSize)++;
   (*dataPacket) = (unsigned char*)realloc((*dataPacket), (*currentSize)
* sizeof(unsigned char));
   (*dataPacket)[(*currentSize) - 1] = 0x00;

   // L1
   (*currentSize)++;
   (*dataPacket) = (unsigned char*)realloc((*dataPacket), (*currentSize)
* sizeof(unsigned char));
   (*dataPacket)[(*currentSize) - 1] = 0x00;

   // Need to subdivide the file into smaller parts (Each packet has
data with partitionSize bytes)
   for (int i = 0; i < partitionSize; i++) {
```

```
        int readBytes = read((*fd), &byte, 1);
        if (readBytes == -1) return -1;
        if (readBytes == 0) break;

        accumulatorOfBytesRead++;
        (*currentSize)++;
        (*dataPacket) = (unsigned char*)realloc((*dataPacket),
(*currentSize) * sizeof(unsigned char));
        (*dataPacket)[(*currentSize) - 1] = byte;
    }

    // K = 256 * L2 + L1
    int L2 = accumulatorOfBytesRead / 256;
    int L1 = accumulatorOfBytesRead - (L2 * 256);

    if (L2 == 0 && L1 == 0) return 0;

    (*dataPacket)[2] = L2;
    (*dataPacket)[3] = L1;

    return 1;
}


/**
* This function only exits when a successful write is done.
* This means that it will only return if we are able to write the full
data.
* packet - packet to be sent
* sizeOfPacket - size of packet to be sent
* returns number of bytes written on success
*         -1 on error
*/
int llwriteWrapper(unsigned char* packet, int sizeOfPacket) {
    int bytesWritten = llwrite(packet, sizeOfPacket);
    if (bytesWritten == -1) return -1;
    return bytesWritten;
}


/**
* Main application function for transmitter.
* linkStruct - struct that contains information about the transmitter
* filename - name of the file to be sent
* returns 0 on success
*         -1 on error
*/
int txApplication(LinkLayer linkStruct, const char* filename) {
    // Open file
    int fd = open((const char *)filename, O_RDONLY);
    if (fd < 0) {
        printf("Unable to open file.\n");
        return -1;
    }

    // Get information about the file
```

```
    struct stat st;
    if (stat(filename, &st) == -1) {
        printf("Unable to get information about the file.\n");
        return -1;
    }
    long fileSize = st.st_size;

    // Open the connection
    if (llopen(linkStruct) != 1) {
        printf("%s: An error occurred inside llopen.\n", __func__);
        return -1;
    }

    // Create the initial control packet
    unsigned char* controlPacket = (unsigned char*)malloc(sizeof(unsigned
char));
    int sizeOfControlPacket = 1;
    controlPacket = createControlPacket(controlPacket,
&sizeOfControlPacket, CSTART, fileSize, (const unsigned char*)filename);
    if (controlPacket == NULL) {
        printf("%s: An error occurred while trying to create the Control
Packet.\n", __func__);
        return -1;
    }

    // Send the start control packet
    int bytesWritten;
    if ((bytesWritten = llwriteWrapper(controlPacket,
sizeOfControlPacket)) == -1) {
        printf("%s: An error occurred while trying to send the END
Control Packet.\n", __func__);
        return -1;
    }

    if (bytesWritten == 0) {
        return 0;
    }

    // Create data packet
    int shouldCreateDataPacket = TRUE;
    while (shouldCreateDataPacket) {
        unsigned char* dataPacket = (unsigned
char*)malloc(sizeof(unsigned char));
        int sizeOfDataPacket = 1;
        shouldCreateDataPacket = createDataPacket(&dataPacket,
&sizeOfDataPacket, &fd);

        if (shouldCreateDataPacket == 0) { // Nothing left to send
            free(dataPacket);
            break;
        }

        if (dataPacket == NULL || shouldCreateDataPacket == -1) {
            printf("%s: An error occurred while trying to create the Data
Packet\n", __func__);
            return -1;
```

```
        }

        // Send the data packet
        if ((bytesWritten = llwriteWrapper(dataPacket, sizeOfDataPacket))
== -1) {
            printf("%s: An error occurred while trying to send the START
Control Packet.\n", __func__);
            return -1;
        }


        if (bytesWritten == 0) {
            return 0;
        }

        sequenceNumber = sequenceNumber == (unsigned char)99 ? 0 :
sequenceNumber + 1;

        free(dataPacket);
    }

    // Create the the end control packet
    controlPacket = createControlPacket(controlPacket,
&sizeOfControlPacket, CEND, fileSize, filename);
    if (controlPacket == NULL) {
        printf("%s: An error occurred while trying to create the END
Control Packet.\n", __func__);
        return -1;
    }

    // Send the end control packet.
    if ((bytesWritten = llwriteWrapper(controlPacket,
sizeOfControlPacket)) == -1) {
        printf("%s: An error occurred while trying to send the END
Control Packet.\n", __func__);
        return -1;
    }

    if (bytesWritten == 0) {
        return 0;
    }

    free(controlPacket);

    // Close connection
    if (llclose(TRUE) == -1) {
        printf("%s: An error occurred in llclose.\n", __func__);
        return -1;
    }

    return 0;
}


/**
* Reads and Checks control packets.
```

```
* fileSize - size of the file to be read
* returns 0 on success
*         -1 on error
*/
int readControlPacket(unsigned char* controlPacket, long* fileSize,
unsigned char* filename, int type) {
   if (type != CEND) {
       int bytesRead = llread(controlPacket);
       if (bytesRead == -1){
           printf("%s: Error in llread\n", __func__);
           return -1;
       }
   }

   // Check if the control packet is correct
   if ((controlPacket[0]) != type) {
       printf("%s: Error in controlPacketType\n", __func__);
       return -1;
   }

   // Check Filesize
   // Type
   if ((controlPacket[1]) != 0) {
           printf("%s: Error in Type of TLV.\n", __func__);
           return -1; // We are expecting a filesize...
   }

   // Length
   char readTLVmax = 4;
   char byteData[4];
   if (controlPacket[2] != (unsigned char)readTLVmax) {
       printf("%s: The length value for filesize is invalid.\n",
__func__);
       return -1;
   }

   int offset = 3;

   // Value
   for (int i = 0; i < readTLVmax; i++) {
       byteData[i] = controlPacket[3 + i];
       offset++;
   }

   (*fileSize) = 0;
   (*fileSize) |= (unsigned char)byteData[0] << 24;
   (*fileSize) |= (unsigned char)byteData[1] << 16;
   (*fileSize) |= (unsigned char)byteData[2] << 8;
   (*fileSize) |= (unsigned char)byteData[3];

   // Check Filename
   // Type
   if (controlPacket[offset] != 1) {
           printf("%s: Error in Type of TLV.\n", __func__);
           return -1; // We are expecting a filesize...
   }
```

```
    // Length
    offset++;
    int filenameSize = controlPacket[offset];

    filename = (unsigned char*)realloc(filename, filenameSize *
sizeof(unsigned char));

    if (filenameSize < 1) {
        printf("%s: Error in controlPacket, filenameSize is less than
one\n", __func__);
        return -1;
    }

    // Value
    offset++;
    for (int i = 0; i < filenameSize; i++) {
        filename[i] = controlPacket[offset + i];
    }
    filename[filenameSize] = '\0';

    return 0;
}


/**
* Reads, checks a data packet and writes contents to a new file.
* fd - file descriptor of the new file
* fileSize - size of the new file
* fileName - name of the received file
* returns number of bytes read on success
*         -1 on error
*/
int readDataPacket(int* fd, long* fileSize, unsigned char* fileName) {
    int continueReadingBytes = 1;
    long totalAmountRead = 0;
    while (continueReadingBytes){
        unsigned char* dataPacket = (unsigned
char*)malloc(MAX_PAYLOAD_SIZE * sizeof(unsigned char));
        int readBytes = 0;

        readBytes = llread(dataPacket);

        if (readBytes == 0) continue; // Nothing left to read
        if (readBytes == -1) {
            printf("%s: An error occurred in llread.\n", __func__);
            return -1;
        }

        if (dataPacket[0] == CEND){
            if (readControlPacket(dataPacket, fileSize, fileName, CEND)
!= 0) {
                printf("%s: Error in readControlPacket.\n", __func__);
                return -1;
            }
            return 0;
```

```
        }

        totalAmountRead += readBytes - 4; // Remove the bytes for header.
        // Sequence number check.
        if (dataPacket[1] != sequenceNumber){
            printf("%s: Unknown error occurred, malformed data packet,
sequence number invalid\n", __func__);
            return -1;
        }

        sequenceNumber = sequenceNumber == (unsigned char)99 ? 0 :
sequenceNumber + 1;

        int l1 = dataPacket[3];
        int l2 = dataPacket[2];
        int k = 256 * l2 + l1;

        if (k == 0 && dataPacket[0] == CDATA) return 0;

        int bytesWritten = 0;
        bytesWritten = write((*fd), dataPacket + 4, k);

        if (bytesWritten == -1) {
            printf("%s: An error occurred while writing to the file.\n",
__func__);
            return -1;
        }

        free(dataPacket);
    }
    return totalAmountRead;
}


/**
* Main application function for receiver.
* linkStruct - struct that contains information about the receiver
* returns 0 on success
*         -1 on error
*/
int rxApplication(LinkLayer linkStruct, const char* filename) {
    // Open the connection
    if (llopen(linkStruct) != 1) {
        printf("%s: An error occurred inside llopen.\n", __func__);
        return -1;
    }

    // Read the start control packet
    unsigned char* controlPacket = (unsigned
char*)malloc(MAX_PAYLOAD_SIZE * sizeof(unsigned char));
    long fileSize;
    unsigned char* txFileName = (unsigned char*)malloc(sizeof(unsigned
char));
    if (readControlPacket(controlPacket, &fileSize, txFileName, CSTART)
!= 0) {
        printf("%s: Error in readControlPacket.\n", __func__);
```

```
        return -1;
    }

    printf("Tx is reading a file with name: %s\n", txFileName);

    // Create file
    int fd = open(filename, O_WRONLY | O_APPEND | O_CREAT, 777);
    if (fd < 0) {
        printf("Unable to open file.\n");
        return -1;
    }

    // Read data packets
    if (readDataPacket(&fd, &fileSize, txFileName) < 0) {
        printf("%s: Error while reading data packet.\n", __func__);
        return -1;
    }

    free(controlPacket);
    free(txFileName);

    // Close the connection
    if (llclose(TRUE) != 1){
        printf("%s: An error ocurred inside llclose.\n", __func__);
        return -1;
    }

    return 0;
}


/**
* Main application layer function, that calls different functions
depending on role.
* serialPort - serial port path
* role - tx or rx
* baudRate - speed of the transmission
* nTries - number of retransmissions in case of failure
* timeout - timer value for timeouts
* filename - name of the file to be sent
*/
void applicationLayer(const char *serialPort, const char *role, int
baudRate, int nTries, int timeout, const char *filename) {
    LinkLayerRole appRole = LlRx;
    if (strcmp("tx", role) == 0) appRole = LlTx;
    else if (strcmp("rx", role) == 0) appRole = LlRx;
    else {
        printf("Please specify a valid role.\n");
        return;
    }

    // Creating the struct
    LinkLayer linkStruct = {
        .role = appRole,
        .baudRate = baudRate,
        .nRetransmissions = nTries,
```

```
        .timeout = timeout
    };
    strcpy(linkStruct.serialPort, serialPort);

    // Call function depending on role
    if (appRole == LlTx) {
        if (txApplication((linkStruct), filename) == -1) {
            printf("%s, Error in txApplication.\n", __func__);
        }
    } else {
        if (rxApplication((linkStruct), filename) == -1){
            printf("%s, Error in rxApplication.\n", __func__);
        }
    }
}
```

Appendix III - link_layer.*h*

```
// 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_
```

Appendix IV - *link_layer.c*

```
// Link layer protocol implementation
#include "link_layer.h"
#include "serial_port.h"

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

// MISC
#define _POSIX_SOURCE 1 // POSIX compliant source
#define FALSE 0
#define TRUE 1

volatile int STOP = FALSE;

// Buffers
#define BUF_SIZE 256

// I frame number
#define I_FRAME_0 0x00
#define I_FRAME_1 0x80

// Frame Fields
#define FLAG 0x7E

#define ADDRESS_SENT_BY_TX 0x03 // or replies sent by receiver.
#define ADDRESS_SENT_BY_RX 0x01 // or replies sent by transmitter.

#define CONTROL_SET 0x03
#define CONTROL_UA 0x07

#define CONTROL_RR0 0xAA
#define CONTROL_RR1 0xAB
#define CONTROL_REJ0 0x54
#define CONTROL_REJ1 0x55
#define CONTROL_DISC 0x0B
```

```c
#define ESCAPE_OCTET 0x7D
#define ESCAPE_XOR 0x20

// Reader State Machine && Acknowledgement State Machine
typedef enum {START, FLAG_RCV, A_RCV, C_RCV, BCC_OK, STOP_STATE,
CHECK_DATA} state_t;

// Serial Port (File Descriptor)
static int fd;
static int numberOfRetransmitions = 0;
static int timeout = 0;
static LinkLayerRole role;

// Previous C Field (for rx)
static int prevCField = 1;

// C Field to send next (for tx)
static int CFieldToSendNext = 0;

// Stastics
unsigned long totalNumOfFrames = 0;
unsigned long totalNumOfValidFrames = 0;
unsigned long totalNumOfInvalidFrames = 0;
unsigned long totalNumOfDuplicateFrames = 0;
unsigned long totalNumOfRetransmissions = 0;
unsigned long totalNumOfTimeouts = 0;


// Handler
int alarmEnabled = FALSE;
int alarmCount = 0;

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


/**
* Supervision frames and Unnumbered frames state machine
* controlField - control field to be checked (depending on the frame
type)
* ringringEnabled - Flag (because both tx and rx use this function)
* returns 0 on success
*         -1 on error
*/
int checkSUFrame(char controlField, int* ringringEnabled){
   state_t state = START;
   while (state != STOP_STATE && (*ringringEnabled)) {
       unsigned char byte = 0;
       int rb = 0;
       if ((rb = readByte(&byte)) == -1) {
           printf("%s: An error occurred inside readByte.\n", __func__);
```

```
            return -1;
        }
        if (rb == 0) continue;
        unsigned char BCC1 = 0x00;

        switch (state) {
            case START:
                if (byte == FLAG) state = FLAG_RCV;
                break;
            case FLAG_RCV:
                state = byte == FLAG ? FLAG_RCV : (byte ==
ADDRESS_SENT_BY_TX ? A_RCV : START);
                break;
            case A_RCV:
                state = byte == FLAG ? FLAG_RCV : (byte == controlField ?
C_RCV : START);
                break;
            case C_RCV:
                BCC1 = ADDRESS_SENT_BY_TX ^ controlField;
                state = byte == FLAG ? FLAG_RCV : (byte == BCC1 ? BCC_OK
: START);
                break;
            case BCC_OK:
                state = byte == FLAG ? STOP_STATE : START;
                break;
            case STOP_STATE:
                break;
            default:
                state = START;
                break;
        }
    }
    return 0;
}


////////////////////////////////////////////////
// LLOPEN
////////////////////////////////////////////////
/**
* Function that opens the connection between tx and rx
* connectionParameters - connection parameters (about tx or rx)
* returns 1 on success
*         -1 on error
*/
int llopen(LinkLayer connectionParameters) {
    if (signal(SIGALRM, alarmHandler) == SIG_ERR) {
        printf("%s: An error occurred inside signal.\n", __func__);
        return -1;
    }
    numberOfRetransmitions = connectionParameters.nRetransmissions;
    timeout = connectionParameters.timeout;
    role = connectionParameters.role;

    if ((fd = openSerialPort(connectionParameters.serialPort,
connectionParameters.baudRate)) < 0) {
```

```
        return -1;
    }

    if (role == LlTx) { // Transmitter
        while (alarmCount < numberOfRetransmitions) {
            int bytesWritten = 0;
            if (alarmEnabled == FALSE){
                // Assemble SET frame
                unsigned char BCC1 = ADDRESS_SENT_BY_TX ^ CONTROL_SET;
                unsigned char set_array[5] = {FLAG, ADDRESS_SENT_BY_TX,
CONTROL_SET, BCC1, FLAG};

                // Send SET frame
                bytesWritten = writeBytes(set_array, 5);
                if (bytesWritten == -1) {
                    printf("%s: Error in writeBytes.\n", __func__);
                    return -1;
                }

                alarm(timeout); // Set alarm to be triggered after
timeout
                alarmEnabled = TRUE;
            }

            if (bytesWritten == 5) {
                alarm(timeout); // Set alarm to be triggered after
timeout
                alarmEnabled = TRUE;
                int csu = checkSUFrame(CONTROL_UA, &alarmEnabled);
                if (csu == -1) {
                    printf("%s: An error occoures inside
checkSUFrame.\n", __func__);
                    return -1;
                }
                else if (alarmEnabled){
                    alarm(0);
                    alarmEnabled = FALSE;
                    alarmCount = 0;
                    return 1;
                }
            }
            totalNumOfRetransmissions++;
        }
    } else if (role == LlRx) { // Receiver
        int enterCheckSUFrame = TRUE;
        while (enterCheckSUFrame) {
            int csu = checkSUFrame(CONTROL_SET, &enterCheckSUFrame);

            if (csu == -1) {
                printf("%s: An error occurred inside checkSUFrame.\n",
__func__);
                return -1;
            }

            int BCC1 = ADDRESS_SENT_BY_TX ^ CONTROL_UA;
```

```
            unsigned char ua_array[5] = {FLAG, ADDRESS_SENT_BY_TX,
CONTROL_UA, BCC1, FLAG};

            int wb = writeBytes(ua_array, 5);
            if (wb == -1) {
                printf("%s: An error occurred inside writeBytes.\n",
__func__);
                return -1;
            }
            else if (wb == 5) return 1;
            else {
                totalNumOfRetransmissions++;
                continue;
            }
        }
    }
    return -1;
}

////////////////////////////////////////////////
// LLWRITE
////////////////////////////////////////////////
/**
* Reads an I frame response (ACK or NACK) to determine which frame to
send next
* returns 0 on valid frame
*         1 on invalid frame
*        -1 on error
*/
int readIFrameResponse() {
    state_t state = START;
    unsigned char BCC1 = 0x00;
    unsigned char byte;
    int isInvalid = 0;

    while (state != STOP_STATE && alarmEnabled) {
        int rb = readByte(&byte);
        if (rb == -1) return -1;
        if (rb == 0) continue;
        switch (state) {
            case START:
                state = byte == FLAG ? FLAG_RCV : START;
                break;
            case FLAG_RCV:
                state = byte == FLAG ? FLAG_RCV : (byte ==
ADDRESS_SENT_BY_TX ? A_RCV : START);
                break;
            case A_RCV:
                switch (byte) {
                    case CONTROL_RR0:
                        totalNumOfValidFrames++;
                        state = C_RCV;
                        CFieldToSendNext = 0;
                        break;
                    case CONTROL_RR1:
                        totalNumOfValidFrames++;
```

```
                        state = C_RCV;
                        CFieldToSendNext = 1;
                        break;
                    case CONTROL_REJ0:
                        totalNumOfInvalidFrames++;
                        isInvalid = 1;
                        state = C_RCV;
                        break;
                    case CONTROL_REJ1:
                        totalNumOfInvalidFrames++;
                        isInvalid = 1;
                        state = C_RCV;
                        break;
                    case FLAG:
                        state = FLAG_RCV;
                        break;
                    default:
                        state = START;
                        break;
                }
                BCC1 = ADDRESS_SENT_BY_TX ^ byte;
                break;
            case C_RCV:
                state = byte == FLAG ? FLAG_RCV : (byte == BCC1 ? BCC_OK
: START);
                break;
            case BCC_OK:
                state = byte == FLAG ? STOP_STATE : START;
                break;
            case STOP_STATE:
                break;
            default:
                state = START;
                break;
        }
    }
    return isInvalid;
}

/**
* Function that tx uses to write frames to the serial port
* buf - frame to write to serial port (before byte stuffing)
* bufSize - Size of the frame to write to serial port (before byte
stuffing)
* returns number of data bytes written (without byte stuffing) on
success
*         -1 on error
*/
int llwrite(const unsigned char *buf, int bufSize) {
    if (bufSize < 0 || buf == NULL) return -1;

    int newFrameSize = bufSize + 6;
    int numBytesStuffed = 0;
    for (int i = 0; i < bufSize; i++) { // Get the new frame size for
byte stuffing
        if (buf[i] == FLAG || buf[i] == ESCAPE_OCTET) {
```

```
            numBytesStuffed++;
            newFrameSize++;
        }
    }

    unsigned char* frame = (unsigned char*)malloc(sizeof(unsigned char) *
(newFrameSize));

    frame[0] = FLAG;
    frame[1] = ADDRESS_SENT_BY_TX;

    if (CFieldToSendNext) frame[2] = 0x80; // Send frame 1 next
    else frame[2] = 0x00; // Send frame 0 next

    frame[3] = frame[1] ^ frame[2];

    // Byte Stuffing
    int j = 0;
    for (int i = 0; i < bufSize; i++) {
        if (buf[i] == FLAG || buf[i] == ESCAPE_OCTET){
            frame[4 + j] = ESCAPE_OCTET;
            j++;
            frame[4 + j] = buf[i] ^ ESCAPE_XOR; // Do the XOR
        } else {
            frame[4 + j] = buf[i];
        }
        j++;
    }

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

    // BCC2 byte stuffing
    if (BCC2 == FLAG || BCC2 == ESCAPE_OCTET) {
        frame = (unsigned char*)realloc(frame, newFrameSize +
sizeof(unsigned char) * 2);
        newFrameSize++;
        frame[newFrameSize - 3] = ESCAPE_OCTET;
        frame[newFrameSize - 2] = BCC2 ^ ESCAPE_XOR;
        numBytesStuffed++;
    } else {
        frame[newFrameSize - 2] = BCC2;
    }

    frame[newFrameSize - 1] = FLAG;

    int wb = 0;

    if (signal(SIGALRM, alarmHandler) == SIG_ERR) {
        printf("%s: An error occurred inside signal.\n", __func__);
        return -1;
    }
    alarm(0);
```

```
    alarmCount = 0;
    alarmEnabled = FALSE;

    int previousCFieldToSendNext = CFieldToSendNext;

    while (alarmCount < numberOfRetransmitions) {
        if (alarmEnabled == FALSE){
            int bytesWritten = writeBytes(frame, newFrameSize);
            totalNumOfFrames++;

            if (bytesWritten == -1) {
                printf("%s: An error occurred inside writeBytes.\n",
__func__);
                return -1;
            } else {
                int response = 0;
                alarm(timeout); // Set alarm
                alarmEnabled = TRUE;
                if ((response = readIFrameResponse()) == -1) {
                    printf("%s: An error occured in
readIFrameResponse.\n", __func__);
                    return -1;
                }

                if (response == 1) { // REJ
                    alarm(0);
                    alarmEnabled = FALSE;
                    alarmCount = 0;
                }

                if (response == 0 && (previousCFieldToSendNext !=
CFieldToSendNext)) {
                    alarm(0);
                    wb = bytesWritten - 6 - numBytesStuffed;
                    alarmEnabled = FALSE;
                    alarmCount = 0;
                    break;
                }
            }

            totalNumOfRetransmissions++;
        }
    }

    free(frame);

    // Return number of writer characters
    return wb;
}

////////////////////////////////////////////////
// LLREAD
////////////////////////////////////////////////
/**
* Helper function that sends an ACK to Tx.
```

```
* The function should get the values that is inside the frame and use it
to respond to Tx.
*
* This way if Tx sends frame 0, Rx should tell Tx that it wants frame 1.
*
* returns void
*
**/
void sendAck(unsigned char receivedCField) {
   unsigned char prevCFieldChar = prevCField ? I_FRAME_1 : I_FRAME_0; //
If previous C Field is 1, then previous C Field Char is I_FRAME_1
(0x80).
   unsigned char RR = 0x00;
   if (receivedCField == prevCFieldChar){ // Frame is duplicate
       RR = prevCField ? CONTROL_RR0 : CONTROL_RR1;
   }
   else {
       if (receivedCField == I_FRAME_0) {
           RR = CONTROL_RR1;
           prevCField = 0;
       }
       else {
           RR = CONTROL_RR0;
           prevCField = 1;
       }
   }

   // Conditions
   unsigned char BCC1 = RR ^ ADDRESS_SENT_BY_TX;

   // Send ACK
   unsigned char ua_array[5] = {FLAG, ADDRESS_SENT_BY_TX, RR, BCC1,
FLAG};
   if (writeBytes(ua_array, 5) == -1) {
       printf("%s: An error occurred in writeBytes\n", __func__);
   }
}

/**
* Function that rx uses to read frames from the serial port
* packet - buffer to read the frame data into
* returns number of data bytes read on success
*        -1 on error
**/
int llread(unsigned char *packet) {
   if (packet == NULL) {
       printf("%s: An error occurred, packet is NULL\n", __func__);
       return -1;
   }

   int state = START;
   unsigned char* dataFrame = (unsigned char *)malloc(sizeof(unsigned
char)); // The data from the information frame will be stored here.
   int currentDataFrameIt = 0;
   unsigned char receivedCField = 0x00;
```

```
   while (state != STOP_STATE) {
       unsigned char byte = 0;
       int rb = 0;
       if ((rb = readByte(&byte)) == -1) {
           printf("%s: An error occurred in readByte.\n", __func__);
           return -1;
       }
       if (rb == 0) continue;

       unsigned char BCC1 = 0x00;

       switch (state) {
           case START:
               if (byte == FLAG) state = FLAG_RCV;
               break;
           case FLAG_RCV:
               state = byte == FLAG ? FLAG_RCV : (byte ==
ADDRESS_SENT_BY_TX ? A_RCV : START);
               break;
           case A_RCV:
               receivedCField = byte;
               if (byte == FLAG) state = FLAG_RCV;
               else if ((byte == I_FRAME_0) || (byte == I_FRAME_1))
state = C_RCV;
               else state = START;
               break;
           case C_RCV:
               BCC1 = ADDRESS_SENT_BY_TX ^ receivedCField;
               state = byte == FLAG ? FLAG_RCV : (byte == BCC1 ? BCC_OK
: START);
               break;
           case BCC_OK:
               dataFrame[currentDataFrameIt] = byte;
               currentDataFrameIt++;
               dataFrame = (unsigned char*)realloc(dataFrame,
(currentDataFrameIt + 1) * sizeof(unsigned char));

               if (dataFrame == NULL) {
                   printf("%s: An error occurred while doing realloc,
dataFrame is NULL.\n", __func__);
                   return -1;
               }

               // When byte is equal to flag -> Stop reading data and go
check the data we received.
               if (byte == FLAG){
                   state = CHECK_DATA;
               }
       }

       if (state == CHECK_DATA) {
           int data_bcc2_flag_size = currentDataFrameIt;
           unsigned char* actualData = (unsigned
char*)malloc(sizeof(unsigned char));
           int actualDataIt = 0;
           int sizeOfActualData = 1;
```

```
            int expectDestuffing = FALSE;
            for (int i = 0; i < (data_bcc2_flag_size - 1); i++) { // Byte
destuffing

                if (actualDataIt != 0) {
                    sizeOfActualData++;
                    actualData = (unsigned char*)realloc(actualData,
sizeOfActualData * sizeof(unsigned char));
                }

                if (dataFrame[i] != ESCAPE_OCTET) {
                    actualData[actualDataIt] = dataFrame[i];
                    actualDataIt++;
                } else {
                    // We know that the current byte is not to be added
                    // And we take the next byte, xor it and add it to
the actualData
                    if (i + 1 > (data_bcc2_flag_size - 1)) {
                        printf("%s: An error occurred inside the byte
destuffing.\n", __func__);
                        return -1;
                    }
                    i++;
                    actualData[actualDataIt] = (dataFrame[i] ^
ESCAPE_XOR);
                    actualDataIt++;
                }

            }

            // Check BCC2
            unsigned char dataAccm = 0x00;

            for (int i = 0; i < sizeOfActualData-1; i++) {
                dataAccm ^= actualData[i];  // EXOR all the destuffed
data bytes
            }

            // Case - XOR is invalid or the data is too big (Reject)
            if (dataAccm != actualData[actualDataIt-1] ||
(sizeOfActualData-1) > MAX_PAYLOAD_SIZE) { // If dataAccm is not the
same as BCC2, something went wrong

                unsigned char REJ = 0x00;
                if (prevCField == 0) REJ = CONTROL_REJ0;
                else REJ = CONTROL_REJ1;

                unsigned char BCC1 = REJ ^ ADDRESS_SENT_BY_TX;

                // SEND NACK
                unsigned char ua_array[5] = {FLAG, ADDRESS_SENT_BY_TX,
REJ, BCC1, FLAG};
                if (writeBytes(ua_array, 5) == -1){
                    printf("%s: An error occurred in writeBytes\n",
__func__);
                    return -1;
```

```
                }
                state = START;

                // Clean allocated space
                free(dataFrame);
                dataAccm = 0;
                currentDataFrameIt = 0;
                dataFrame = (unsigned char *)malloc(sizeof(unsigned
char)); // The data from the information frame will be stored here.

                free(actualData);
                actualData = (unsigned char*)malloc(sizeof(unsigned
char));
                actualDataIt = 0;
                sizeOfActualData = 1;
                totalNumOfFrames++;
                totalNumOfInvalidFrames++;
            } else {
                unsigned char prevCFieldChar = prevCField ? I_FRAME_1 :
I_FRAME_0;
                // Case - Frame is a duplicate (Accept and discard)
                if (prevCFieldChar == receivedCField){
                    sendAck(receivedCField);
                    free(dataFrame);
                    free(actualData);
                    totalNumOfFrames++;
                    totalNumOfDuplicateFrames++;
                    return 0;
                }

                // Case - Frame accepted (Accept)
                for (int i = 0; i < sizeOfActualData; i++) {
                    packet[i] = (unsigned char)actualData[i];
                }

                sendAck(receivedCField);
                totalNumOfFrames++;
                totalNumOfValidFrames++;
                return sizeOfActualData;
            }
        }
    }

    printf("%s: An error occurred.\n", __func__);
    return -1;
}

////////////////////////////////////////////////
// LLCLOSE
////////////////////////////////////////////////
int llclose(int showStatistics) {
    if (showStatistics) printf("Statistics:\n");
    if (role == LlTx) { // Transmitter
        while (alarmCount < numberOfRetransmitions) {
            int bytesWritten = 0;
```

```
            if (alarmEnabled == FALSE){
                alarm(timeout); // Set alarm to be triggered after
timeout
                alarmEnabled = TRUE;

                // Assemble DISC frame
                int array_size = 5;
                unsigned char BCC1 = ADDRESS_SENT_BY_TX ^ CONTROL_DISC;
                unsigned char set_array[5] = {FLAG, ADDRESS_SENT_BY_TX,
CONTROL_DISC, BCC1, FLAG};

                // Send DISC frame
                while (bytesWritten != 5) {
                    bytesWritten = writeBytes((set_array +
sizeof(unsigned char) * bytesWritten), array_size - bytesWritten);
                    if (bytesWritten == -1) {
                        printf("%s: An error occurred inside
writeBytes.\n", __func__);
                        return -1;
                    }
                }
            }
            if (bytesWritten == 5) {
                alarm(0);
                alarmEnabled = FALSE;
                int csu = checkSUFrame(CONTROL_DISC, &alarmEnabled);
                if (csu == -1) {
                    printf("%s: An error occurred inside
checkSUFrame.\n", __func__);
                    return -1;
                }
                else break;
            }
            totalNumOfRetransmissions++;
        }
        if (showStatistics) {
            printf("Number of dropped packets (TX): %d\n",
((int)totalNumOfFrames) - ((int)(totalNumOfValidFrames)) -
((int)(totalNumOfInvalidFrames)));
            printf("Total number of frames that were retransmitted:
%ld\n", totalNumOfRetransmissions);
            printf("Total number of timeouts: %ld\n",
totalNumOfTimeouts);
        }
    } else if (role == LlRx) { // Receiver
        int enterCheckSUFrame = TRUE;
        while (enterCheckSUFrame) {

            int csu = checkSUFrame(CONTROL_DISC, &enterCheckSUFrame);
            if (csu == -1) {
                printf("%s: An error occurred inside checkSUFrame.\n",
__func__);
                return -1;
            }

            int BCC1 = ADDRESS_SENT_BY_TX ^ CONTROL_DISC;
```

```
            unsigned char ua_array[5] = {FLAG, ADDRESS_SENT_BY_TX,
CONTROL_DISC, BCC1, FLAG};

            int wb = writeBytes(ua_array, 5);

            if (wb == -1) {
                printf("%s: An error occurred inside writeBytes.\n",
__func__);
                return -1;
            }
            else if (wb == 5) {
                if (showStatistics){
                    printf("Number of dropped packets (RX): %d\n",
((int)totalNumOfFrames) - ((int)(totalNumOfValidFrames)) -
((int)(totalNumOfInvalidFrames)) - ((int)(totalNumOfDuplicateFrames)));
                    printf("Number of frames received that were
duplicate: %ld\n", totalNumOfDuplicateFrames);
                }
                break;
            }
            else continue;
        }
    }

    if (showStatistics){
        printf("Number of frames that were sent/received and are valid:
%ld\n", totalNumOfValidFrames);
        printf("Number of frames that were sent/received and are invalid:
%ld\n", totalNumOfInvalidFrames);
        printf("Total number of frames that were sent/received: %ld\n",
totalNumOfFrames);
    }

    if (closeSerialPort() == -1){
        printf("%s: Error while closing serial port\n", __func__);
        return -1;
    }
    return 1;
}
```

Appendix V - *serial_port.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 for a byte received from the serial port and read it (must
// check whether a byte was actually received from the return value).
// Returns -1 on error, 0 if no byte was received, 1 if a byte was
received.
int readByte(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 writeBytes(const char *bytes, int numBytes);

#endif // _SERIAL_PORT_H_
```

Appendix VI - *serial_port.c*

```
// Serial port interface implementation
// DO NOT CHANGE THIS FILE

#include "serial_port.h"

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

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

int 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(void)
{
    // Restore the old port settings
    if (tcsetattr(fd, TCSANOW, &oldtio) == -1)
    {
        perror("tcsetattr");
        return -1;
    }

    return close(fd);
}


// Wait for a byte received from the serial port and read it (must
// check whether a byte was actually received from the return value).
// Returns -1 on error, 0 if no byte was received, 1 if a byte was
received.
int readByte(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 writeBytes(const char *bytes, int numBytes)
{
    return write(fd, bytes, numBytes);
}
```

Appendix VII - *main.c*

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

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

#include "application_layer.h"

#define N_TRIES 3
```

```c
#define TIMEOUT 4


// Arguments:
//   $1: /dev/ttySxx
//   $2: baud rate
//   $3: tx | rx
//   $4: filename
int main(int argc, char *argv[])
{
   if (argc < 5) {
       printf("Usage: %s /dev/ttySxx baudrate tx|rx filename\n",
argv[0]);
       exit(1);
   }

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

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

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

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

```
            TIMEOUT,
            filename);

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

    return 0;
}
```

Appendix VIII - FER Variation

- File size = 10968 bytes = 87744 bits
- Baudrate (C) = 38400 bits/s
- Packet size = 1000 bytes
- R (bits/s) = File size / time
- Efficiency = S = R/C

FER (%)	Time (s)	R (bits/s)	S (%)
0	4,400125	19941,25167	51,9303
2	5,396703	16258,81580	42,3406
4	6,612466	13269,48222	34,5559
6	6,958467	12609,67394	32,8377
10	7,881125	11133,43590	28,9933
20	15,091797	5814,01936	15,1407
40	111,440612	787,36107	2,0504

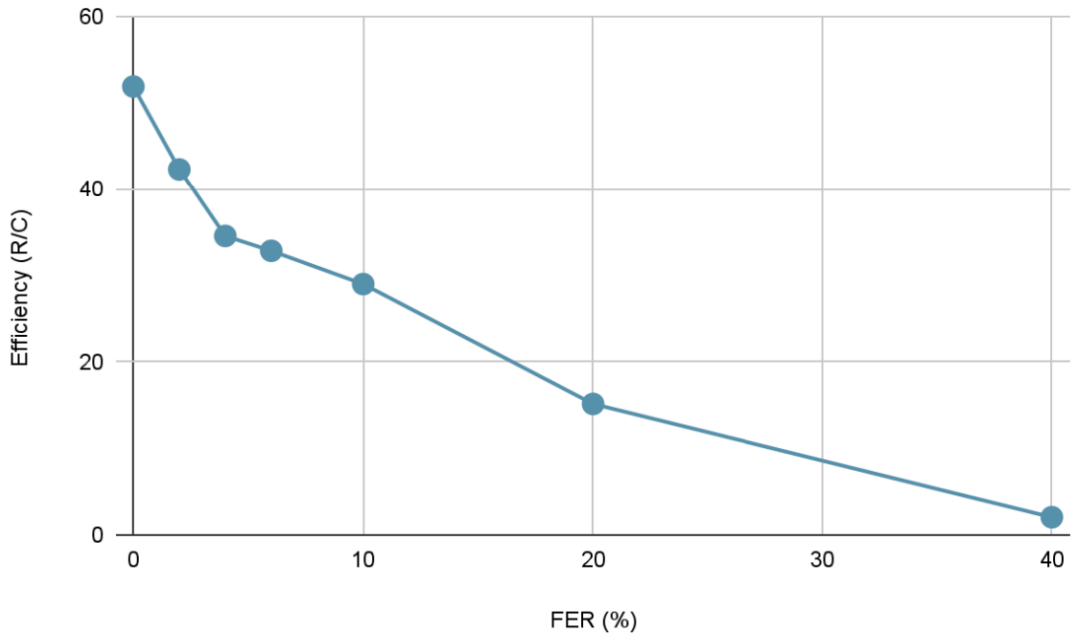

Figure 3: Results of FER variation tests

Appendix IX - Baudrate Variation

- File size = 10968 bytes = 87744 bits
- Packet size = 1000 bytes
- Baudrate = C
- R (bits/s) = File size / time
- Efficiency = S = R/C

Baudrate (bits/s)	Time (s)	R (bits/s)	S (%)
1200	174,43036	503,031697	41,9193
1800	64,920965	1351,38247	75,0768
2400	48,711051	1801,31609	75,0548
4800	25,277380	3471,24583	72,3176
9600	16,987431	5165,23069	53,8045
19200	7,625322	11506,9239	59,9319
38400	4,807885	18250,0205	47,5261
57600	3,793717	23128,7679	40,1541
115200	2,792596	31420,2269	27,2745

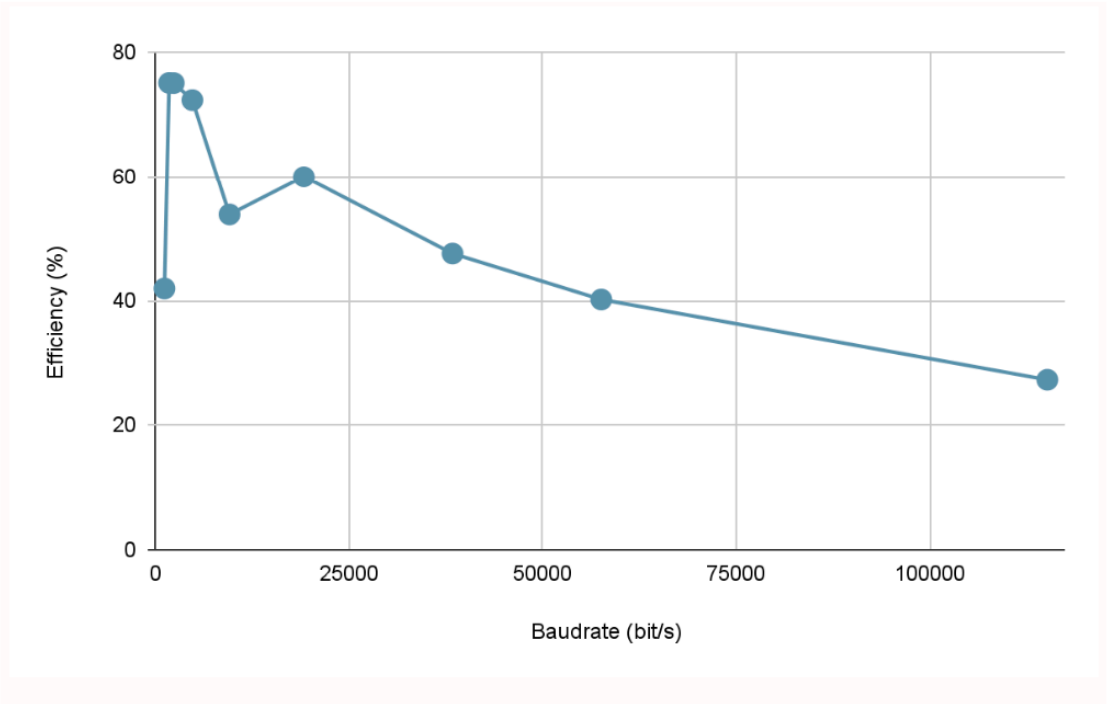

Figure 4: Results of baudrate variation tests

Appendix X - Frame Size Variation

- File size = 10968 bytes = 87744 bits
- Packet size = 1000 bytes
- Baudrate (C) = 38400
- R (bits/s) = File size / time
- Efficiency = S = R/C

Frame Size (bytes)	Time (s)	R (bits/s)	S (%)
50	6,326935	13868,3265	36,1154
100	5,498403	15958,0882	41,5575
200	5,972235	14691,9872	38,2604
500	5,096384	17216,9130	44,8357
800	5,135961	17084,2419	44,4902
1000	4,674305	18771,5607	48,8842

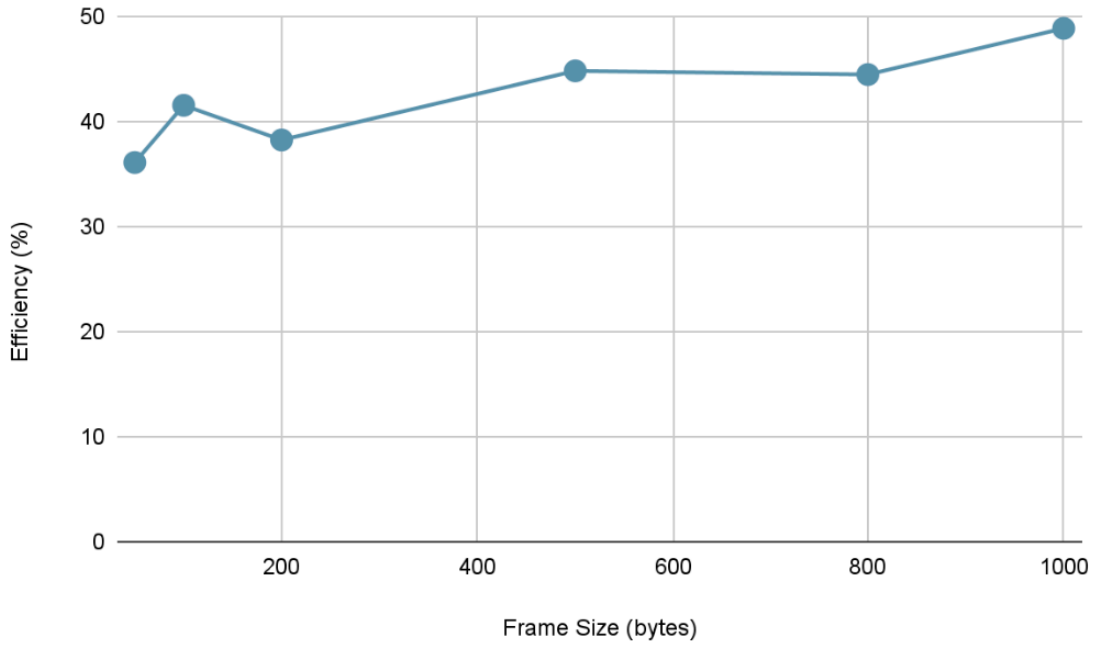

Figure 5: Results of frame size variation tests