# Data Link Protocol

*1st Lab Work*

**Computer Networks**
**3LEIC04**

Rui Carvalho (up202108807@edu.fe.up.pt )
Juan Ignacio Medone Sabatini (up202302380@edu.fe.up.pt)

Porto, December 2nd, 2023

## Summary

This work, developed in the context of a Computer Networks course, has as its purpose the implementation of a data link protocol in order to transmit files using the Serial Port RS-232.

By developing this project we were able to make use of the material taught in theoretical classes to implement the protocol in question, thus consolidating the functioning of the Stop-and-Wait strategy.

## 1. Introduction

The goal of this project was to develop and test an efficient data link protocol, according to the specifications presented in the guide, in order to transfer files through the Serial Port RS-232. This report is divided in the following sections:

- **Architecture:** Functional blocks and used interfaces.
- **Code Structure:** Presentation of the main APIs, structures and functions.
- **Main Use Cases:** Identification of the project's behavior, as well as the sequence of the function calls.
- **Logical Link Protocol:** Logical link behavior and implementation strategies.
- **Application Protocol:** Application layer functioning and implementation strategies.
- **Validation:** Tests carried out to evaluate the implementation preciseness.
- **Data Link Protocol Efficiency:** Measure of the efficiency of the Stop-and-Wait protocol implemented in the data link layer.
- **Conclusions:** Summary of the information presented in the previous sections and final thoughts about the learning goals achieved.

## 2. Architecture

The project was developed based on two main functional blocks: the Link Layer and the Application Layer.

The Link Layer is responsible for the establishment and ending of the connection, the creation and sending of data frames through the serial port and also for the validation of the received frames, including sending error messages in case any problem occurs during the data transmission.

The Application Layer uses the Link Layer API to send and receive data packets that belong to a certain file. Thus, this is considered the nearest layer in relation to the user and there we can define some variables, such as the size of the information frames and the transfer speed, as well as the maximum number of retransmissions (in case of any error occuring during the transmission).

# 3. Code Structure

## Application Layer

In this layer the only implemented function was:

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

## Link Layer

In this layer we made use of two auxiliary data structures: *LinkLayer*, where the data transfer parameters are specified, and *LinkLayerRole*, which identifies if the computer runs as the transmitter or as the receiver.

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

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

The implemented functions were the following:

```
// 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);
```

## Auxiliar Funcs

This was an auxiliary file that we created in order to have the code organized. In this file, we made use of an auxiliary structure: *LinkLayerState*.

```
typedef enum
{
   START_TX,
   FLAG_RCV,
   A_RCV,
   C_RCV,
   BCC1_OK,
   STOP_R,
   DATA_FOUND_ESC,
   READING_DATA,
   DISCONNECTED,
   BCC2_OK
} LinkLayerState;
```

The implemented functions in this file were the following:

```
int linkTx(LinkLayer connection);
int linkRx(LinkLayer connection);
```

```
void alarmHandler(int signal);
int makeConnection(const char* serialPort);
int sendSupervisionFrame(unsigned char A, unsigned char C);
unsigned char* getControlPacket(const unsigned int c, const char*
filename, long int length, unsigned int* size);
unsigned char* getDataPacket(unsigned char sequence, unsigned char *data,
int dataSize, int *packetSize);
unsigned char* parseControlPacket(unsigned char* packet, int size,
unsigned long int *fileSize);
void parseDataPacket(const unsigned char* packet, const unsigned int
packetSize, unsigned char* buffer);
unsigned char* getData(FILE* fd, long int fileLength);
unsigned char readControlFrame();
```

## 4. Main Use Cases

The program execution is done using two terminals, one on each computer, where one takes the role of the transmitter and the other of the receiver. The functions used and the sequence in which they are called will be different.

```
$ make run_<ROLE>
```

- ROLE: ‘tx’ for the transmitter, ‘rx’ for the receiver

Thus, on one hand, the **main functions used by the transmitter** are:

1. ***llopen***, used to establish the connection between the transmitter and the receiver, first setting up the connection with the serial port and then transferring certain control packets (making use of the *makeConnection* and *linkTx* functions).
2. ***getControlPacket***, creates a control packet.
3. ***getData***, returns the content of the file to transfer.
4. ***llwrite***, creates and sends an information frame based on the packet received as argument.
5. ***readControlFrame***, a state machine that reads and validates the receiver’s answer after any kind of writing on the serial port.
6. ***llclose***, used to end the connection between the transmitter and the receiver, through the transfer of control packets.

On the other hand, the **main functions used by the receiver** are:

1. ***llopen***, used in a similar way as the transmitter but with the *linkRx* function.

2. ***llread***, a state machine that manages and validates the reception of data frames and control frames.
3. ***sendSupervisionFrame***, creates and sends a supervision frame based on the frame read on *llread*.
4. ***parseControlPacket***, returns the characteristics of the file being transferred. These characteristics are contained in the control packet in TLV format passed as an argument.
5. ***getDataPacket***, returns a segment of the file using the data packet passed as an argument.

## 5. Logical Link Protocol

The Link layer is the one that interacts directly with the serial port, responsible for communication between the transmitter and the receiver. For this purpose, we use the Stop-and-Wait protocol to establish and terminate the connection and for sending supervision and information frames.

Connection establishment is performed by the function *llopen*. Initially, after opening and configuring the serial port, the transmitter sends a SET supervision frame and waits for the receiver to respond with a UA supervision frame. When the receiver receives the SET, it responds with UA. If the transmitter receives the UA frame, the connection has been successfully established. After establishing the connection, the transmitter begins to send information that will be read by the receiver.

The information is sent by the function *llwrite*. This function receives a control or data packet and applies a byte stuffing strategy to it in order to avoid conflicts with data bytes that are the same as the frame flags. Subsequently, it transforms this packet into an information frame, sends it to the receiver, and waits for its response. If the frame is rejected, the sending process is repeated until it is accepted or exceeds the maximum number of retransmissions. Each transmission attempt has a time limit after which a timeout occurs.

On the other side, the information is read by *llread*. This function reads the information received from the serial port and checks its validity. Initially, it destuffs the data field of the frame and validates BCC1 and BCC2, which check if any errors occurred during the transmission.

In the end, the connection is terminated by the function *llclose*. This is invoked by the transmitter when the number of failed attempts is exceeded or when the transfer of data packets is completed. The transmitter sends a DISC supervision frame and waits for the receiver to respond with the same, terminating its operation. When the transmitter receives DISC again, it responds with UA and ends the connection.

## 6. Application Protocol

The application layer is the one that interacts directly with the file being transferred and the user. It allows defining which file to transfer, through which serial port, the transfer speed, the number of data bytes from the file inserted into each packet, the maximum number of retransmissions, and the maximum waiting time for a response from the receiver. File transfer occurs using the Link Layer API, which translates data packets into information frames.

After the initial handshake between the transmitter and the receiver, the entire content of the file is copied into a local buffer through *getData* and fragmented by the Application Layer according to the number of bytes specified in the argument. The first packet sent by the transmitter contains data in TLV format (Type, Length, Value) created by *getControlPacket*, expressing the size of the file in bytes and its name. On the receiver's side, this packet is unpacked by the *parseControlPacket* function in order to create and allocate the necessary space to receive the file.

Each fragment of the file to be transferred is inserted into a data packet through the *getDataPacket* function and sent through the serial port using *llwrite*. Each transmission is accompanied by a response from the receiver, indicating whether it accepts or rejects the packet sent. In the first case, the transmitter sends the next fragment; in the second case, it resends the same fragment. Each packet is individually evaluated by the receiver through the *llread* function, and *parseDataPacket* extracts the original file fragment from the packet when received correctly.

The connection between the two machines ends when *llclose* is invoked, after the completion of the data packets transfer or when the allowed number of retransmissions is exceeded.

## 7. Validation

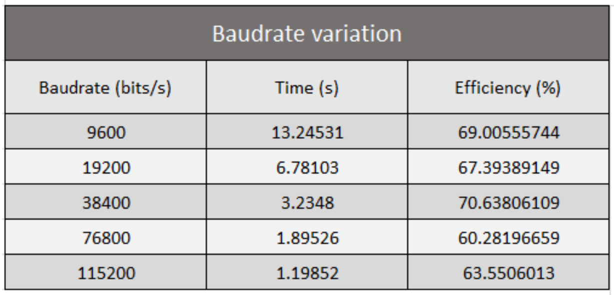
In order to ensure the correct implementation of the developed protocol, the following tests were conducted:

- Transfer of files with different names
- Transfer of files of different sizes
- Partial and total disconnection of the serial port
- Noise injection in the serial port through short-circuiting

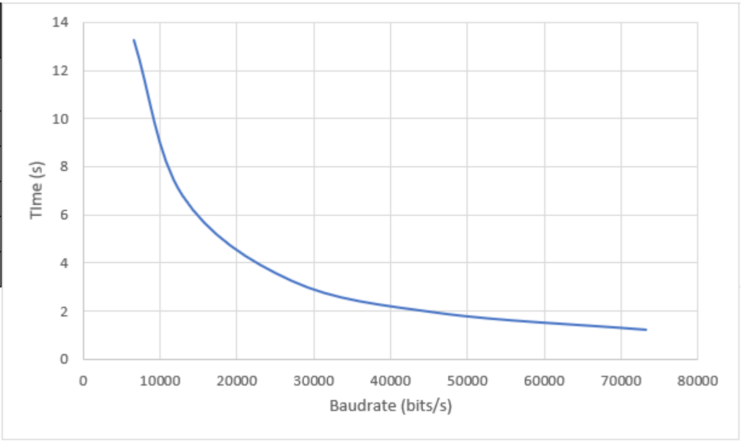
In all the presented scenarios, the implemented Stop-and-Wait protocol ensured the consistency of the transferred file. Most of these tests were also replicated in presence of the instructor during the project presentation in the lab class.

## 8. Data Link Protocol Efficiency

Given the penguin.gif file with a size of 10968 bytes, these are the results obtained when varying the baud rate:



| Baudrate variation | | |
|---|---|---|
| Baudrate (bits/s) | Time (s) | Efficiency (%) |
| 9600 | 13.24531 | 69.00555744 |
| 19200 | 6.78103 | 67.39389149 |
| 38400 | 3.2348 | 70.63806109 |
| 76800 | 1.89526 | 60.28196659 |
| 115200 | 1.19852 | 63.5506013 |



## 9. Conclusions

The Data Link Protocol, consisting of the Link Layer responsible for interacting with the serial port and managing information frames, and the Application Layer that interacts directly with the file being transferred, was very important for consolidating the topics taught in the theoretical classes. Therefore, with this project, we internalized the concepts of byte stuffing, framing, and the operation of the Stop-and-Wait protocol, as well as how it detects errors and deals with them.

# Annex I - Source code

## Main.c

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

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

#include "application_layer.h"

#define BAUDRATE 9600
#define N_TRIES 3
#define TIMEOUT 4

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

    const char *serialPort = argv[1];
    const char *role = argv[2];
    const char *filename = argv[3];
printf("\nStarting 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;
}
```

## 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_
```

## application_layer.c

```c
// Application layer protocol implementation

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

void applicationLayer(const char *serialPort, const char *role, int
baudRate,
                      int nTries, int timeout, const char *filename)
{
    clock_t start_t, end_t;
    double total_t;

    start_t = clock();

    LinkLayerRole enumRole;
    if (strcmp(role, "tx") == 0) {
        enumRole = LlTx;
    } else if (strcmp(role, "rx") == 0) {
        enumRole = LlRx;
    }

    LinkLayer connectionParameters;
    strcpy(connectionParameters.serialPort,serialPort);
    connectionParameters.role = enumRole;
    connectionParameters.baudRate = baudRate;
    connectionParameters.nRetransmissions = nTries;
    connectionParameters.timeout = timeout;

    printf("\nEstablishing connection..\n");
    int openResult = llopen(connectionParameters);
    if (openResult < 0) {
        perror("\nError: failed to establish connection. Terminating
program.\n");
        exit(-1);
    }
    else if (openResult == 1) {
        printf("\nConnection established\n\nTransferring file..\n");
        FILE* file;

        if (enumRole == LlTx) {

            file = fopen(filename,"rb");
            if (file == NULL) {
                perror("File not found\n");
                exit(-1);
            }

            int prev = ftell(file);
            fseek(file, 0L, SEEK_END);
            long int fileSize = ftell(file)-prev;    // file size: 10968
            fseek(file, prev, SEEK_SET);

            // signal the start of the transfer by sending a control
packet
            // cField (values: 2 - startPacket; 3 - endPacket)
            unsigned int controlPacketSize;
            unsigned char *controlPacketStart = getControlPacket(2,
filename, fileSize, &controlPacketSize);    // control packet size: 18
            int startingBytes = llwrite(controlPacketStart,
controlPacketSize);

            if (startingBytes == -1) {
                printf("Error: could not send start packet\n");
                exit(-1);
            }

            unsigned char sequence = 0;
            unsigned char* content = getData(file, fileSize);
            long int bytesLeft = fileSize;

            while (bytesLeft >= 0) {

                int dataSize = bytesLeft > (long int) MAX_PAYLOAD_SIZE ?
MAX_PAYLOAD_SIZE : bytesLeft;
                unsigned char* data = (unsigned char*) malloc(dataSize);
                memcpy(data, content, dataSize);
                int packetSize;
                unsigned char* packet = getDataPacket(sequence, data,
dataSize, &packetSize);

                if(llwrite(packet, packetSize) == -1) {
                    printf("\nError during data packets transmission.
Terminating program.\n");
                    exit(-1);
                }

                bytesLeft -= (long int) MAX_PAYLOAD_SIZE;
                content += dataSize;
                sequence = (sequence + 1) % 255;
            }

            // signal the end of the transfer by sending a control packet
again
            unsigned char *controlPacketEnd = getControlPacket(3,
filename, fileSize, &controlPacketSize);
            int endingBytes = llwrite(controlPacketEnd,
controlPacketSize);

            if (endingBytes == -1) {
                printf("Error: could not send end packet\n");
                exit(-1);
            }

            int showStatistics = FALSE;
            int closeResult = llclose(showStatistics);
            fclose(file);

            if (closeResult == 1)
              printf("File transferred correctly and connection closed
successfuly\n");

            else
              printf("An error occurred while closing the connection\n");
        }
```

```
    else {  // enumRole == LlRx
            unsigned char *packet = (unsigned
char*)malloc(MAX_PAYLOAD_SIZE);
            int packetSize = -1;
            while ((packetSize = llread(packet)) < 0);

            unsigned long int rxFileSize = 0;
            parseControlPacket(packet, packetSize, &rxFileSize);
            FILE* newFile = fopen((char*) filename, "wb+");

            while (TRUE) {
                while ((packetSize = llread(packet)) < 0);
                if (packetSize == 0)
                    break;
                else if (packet[0] != 3) {      // if this is not the
controlPacketEnd, we will process the control packet
                    unsigned char *buffer = (unsigned
char*)malloc(packetSize);
                    parseDataPacket(packet, packetSize, buffer);
                    fwrite(buffer, sizeof(unsigned char), packetSize - 4,
newFile);
                    free(buffer);
                }
                else continue;
            }
            fclose(newFile);
            printf("File transferred correctly and connection closed
successfuly\n");

            end_t = clock();
            total_t = (end_t - start_t) / (float) CLOCKS_PER_SEC;
            printf("Elapsed time: %f sec\n", total_t);
        }
    }
    else {
        printf("An error occurred in the linking process. Terminating the
program\n");
        exit(-1);
    }
}
```

## 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_
```

## link_layer.c

```
// Link layer protocol implementation

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

volatile int STOP = FALSE;
int alarmEnabled = FALSE;
int alarmCount = 0;
int timeout;
int retransmissions = 0;
unsigned char tramaTx = 0;
unsigned char tramaRx = 1;
unsigned char START = 0xFF;
int fd;

// Baudrate settings are defined in <asm/termbits.h>, which is
// included by <termios.h>
#define BAUDRATE B38400
#define _POSIX_SOURCE 1 // POSIX compliant source

#define BUF_SIZE 256

////////////////////////////////////////////////
// LLOPEN
```

```
////////////////////////////////////////////////
int llopen(LinkLayer connectionParameters)
{
    int result;

    fd = makeConnection(connectionParameters.serialPort);
    retransmissions = connectionParameters.nRetransmissions;
    timeout = connectionParameters.timeout;

    if (fd < 0)
      return -1;

    if (connectionParameters.role == LlTx) {
        result = linkTx(connectionParameters);
    }
    else {
        result = linkRx(connectionParameters);
    }

    return result;
}

////////////////////////////////////////////////
// LLWRITE
////////////////////////////////////////////////
int llwrite(const unsigned char *buf, int bufSize)
{
    // prepare I frame to send
    int size_I_buf = bufSize + 6;
    unsigned char* I_buf = (unsigned char *) malloc(size_I_buf);
    I_buf[0] = FLAG;
    I_buf[1] = A_ER;
    I_buf[2] = C_N(tramaTx);
    I_buf[3] = I_buf[1] ^ I_buf[2];
    memcpy(I_buf + 4, buf, bufSize);

    // building BCC2: xor of all D's
    unsigned char BCC2 = buf[0];
    for (unsigned int i = 1 ; i < bufSize ; i++) BCC2 ^= buf[i];
    // data packet
    int j = 4;
    for (unsigned int i = 0 ; i < bufSize ; i++) {
        if (buf[i] == FLAG || buf[i] == ESC) {
            I_buf = realloc(I_buf,++size_I_buf);
            I_buf[j++] = ESC;
        }
        I_buf[j++] = buf[i];
    }
    I_buf[j++] = BCC2;
    I_buf[j++] = FLAG;

    int accepted;
    int rejected;
    alarmCount = 0;

    while (alarmCount < retransmissions) {
      alarm(timeout);
      alarmEnabled = TRUE;
      rejected = 0;
      accepted = 0;

      while (alarmEnabled == TRUE && !rejected && !accepted) {
        write(fd, I_buf, size_I_buf);

        // Wait until all bytes have been written to the serial port
        sleep(1);
        unsigned char result = readControlFrame();

        if (!result)
            continue;

        else if (result == C_REJ(0) || result == C_REJ(1)) // I frame
rejected, needs to read again
            rejected = TRUE;

        else if (result == C_RR(0) || result == C_RR(1)) { // I frame
accepted
            accepted = TRUE;
            tramaTx = (tramaTx+1) % 2;
        }
        else
            continue;
      }

      if (accepted)   // I frame sent correctly
          break;
    }

    free(I_buf);

    if (accepted)
        return size_I_buf;
    else {
        llclose(FALSE);
        return -1;
    }
}

////////////////////////////////////////////////
// LLREAD
////////////////////////////////////////////////
int llread(unsigned char *packet)
{
    LinkLayerState state = START_TX;
    unsigned char buf, cField;
    int i = 0;

    while (state != STOP_R) {
        if (read(fd, &buf, 1) > 0) {
          switch (state) {

              case START_TX:
                  if (buf == FLAG)
                      state = FLAG_RCV;
                  break;

              case FLAG_RCV:
                  if (buf == A_ER)
                      state = A_RCV;
                  else if (buf != FLAG)
                      state = START_TX;
                  break;

              case A_RCV:
                  if (buf == C_N(0) || buf == C_N(1)) {
                      state = C_RCV;
                      cField = buf;
                  }
                  else if (buf == FLAG)
                      state = FLAG_RCV;
                  else if (buf == C_DISC) {
                      sendSupervisionFrame(A_RE, C_DISC);
                      return 0;
                  }
                  else
                      state = START_TX;
                  break;

              case C_RCV:
                  if (buf == (A_ER ^ cField))
                      state = READING_DATA;
                  else if (buf == FLAG)
                      state = FLAG_RCV;
                  else
                      state = START_TX;
                  break;

              case READING_DATA:
                  if (buf == ESC)
                      state = DATA_FOUND_ESC;
                  else if (buf == FLAG) {
                      unsigned char bcc2 = packet[i-1];
                      i--;
                      packet[i] = '\0';
                      unsigned char acc = packet[0];

                      for (unsigned int j = 1; j < i; j++)
                          acc ^= packet[j];
                      if (bcc2 == acc) {
                          state = STOP_R;
                          sendSupervisionFrame(A_RE, C_RR(tramaRx));
                          tramaRx = (tramaRx + 1) % 2;
                          return i;   // amount of bytes read
                      }
                      else {
                          printf("An error ocurred, sending RREJ\n");
                          sendSupervisionFrame(A_RE, C_REJ(tramaRx));
                          return -1;
                      };

                  }
                    else
                        packet[i++] = buf;
                    break;

              case DATA_FOUND_ESC:
                  state = READING_DATA;
                  if (buf == ESC || buf == FLAG)
                      packet[i++] = buf;
                  else {
                      packet[i++] = ESC;
                      packet[i++] = buf;
                  }
                  break;
              default:
                  break;
          }
        }

    }
    return -1;
}
```

```c
////////////////////////////////////////////////
// LLCLOSE
////////////////////////////////////////////////
int llclose(int showStatistics)
{
    unsigned char byte;
    LinkLayerState state = START_TX;

    (void) signal(SIGALRM, alarmHandler);

    while (alarmCount < retransmissions && state != STOP_R)
    {
        sendSupervisionFrame(A_DISC, C_DISC);
        alarm(timeout);
        alarmEnabled = FALSE;

        sleep(1);

        while (alarmEnabled == FALSE && state != STOP_R) {

            if (read(fd, &byte, 1)) {
                switch (state) {
                    case START_TX:
                        if (byte == FLAG) state = FLAG_RCV;
                        break;
                    case FLAG_RCV:
                        if (byte == A_RE) state = A_RCV;
                        else if (byte != FLAG) state = START_TX;
                        break;
                    case A_RCV:
                        if (byte == C_DISC) state = C_RCV;
                        else if (byte == FLAG) state = FLAG_RCV;
                        else state = START_TX;
                        break;
                    case C_RCV:
                        if (byte == (A_RE ^ C_DISC)) state = BCC1_OK;
                        else if (byte == FLAG) state = FLAG_RCV;
                        else state = START_TX;
                        break;
                    case BCC1_OK:
                        if (byte == FLAG) state = STOP_R;
                        else state = START_TX;
                        break;
                    default:
                        break;
                }
            }
        }
        alarmCount++;
    }

    if (state != STOP_R) return -1;

    // send the UA buffer to the receiver
    sendSupervisionFrame(0x03, C_UA);
    sleep(1);

    close(fd);
    return 1;
}
```

## auxiliar_funcs.h

```
#ifndef _AUX_FUNCS_H_
#define _AUX_FUNCS_H_

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

```c
#define FLAG        0x7E
#define A_SET       0x03
#define C_SET       0x03
#define BCC1_SET    (A_SET ^ C_SET)
#define A_UA        0x01
#define C_UA        0x07
#define BCC1_UA     (A_UA ^ C_UA)
#define A_ER        0x03
#define A_RE        0x01
#define A_DISC      0x03
#define C_DISC      0x0B
#define BCC1_DISC   (A_DISC ^ C_DISC)
#define C_N(Ns)     (Ns << 6)
#define ESC         0x7D
#define C_RR(Nr)    ((Nr << 7) | 0x05)
#define C_REJ(Nr)   ((Nr << 7) | 0x01)

typedef enum
{
   START_TX,
   FLAG_RCV,
   A_RCV,
   C_RCV,
   BCC1_OK,
   STOP_R,
   DATA_FOUND_ESC,
   READING_DATA,
   DISCONNECTED,
   BCC2_OK
} LinkLayerState;

int linkTx(LinkLayer connection);
int linkRx(LinkLayer connection);
void alarmHandler(int signal);
int makeConnection(const char* serialPort);
int sendSupervisionFrame(unsigned char A, unsigned char C);
unsigned char* getControlPacket(const unsigned int c, const char*
filename, long int length, unsigned int* size);
unsigned char* getDataPacket(unsigned char sequence, unsigned char *data,
int dataSize, int *packetSize);
```

```
unsigned char* parseControlPacket(unsigned char* packet, int size,
unsigned long int *fileSize);
void parseDataPacket(const unsigned char* packet, const unsigned int
packetSize, unsigned char* buffer);
unsigned char* getData(FILE* fd, long int fileLength);
unsigned char readControlFrame();

#endif
```

## auxiliar_funcs.c

```
#include "link_layer.h"
#include "application_layer.h"
#include "auxiliar_funcs.h"

// Baudrate settings are defined in <asm/termbits.h>, which is
// included by <termios.h>
#define BAUDRATE B38400
#define _POSIX_SOURCE 1 // POSIX compliant source

#define FALSE 0
#define TRUE 1
#define BUF_SIZE 256

extern volatile int STOP;
extern int alarmEnabled;
extern int alarmCount;
extern int timeout;
extern int retransmissions;
extern unsigned char tramaTx;
extern unsigned char tramaRx;
extern unsigned char START;
extern int fd;

int linkTx(LinkLayer connection) {

    // UA buffer that is sent as an answer by the receiver
    unsigned char UA_buffer[1] = {0};

    unsigned char state = START;
    int result = -1;

    (void) signal(SIGALRM, alarmHandler);

    while (alarmCount < connection.nRetransmissions && STOP == FALSE)
    {
        sendSupervisionFrame(A_SET, C_SET);
        sleep(1);

        if (alarmEnabled == FALSE)
        {
            alarm(connection.timeout);      // Sets alarm to be triggered
in 4s
            alarmEnabled = TRUE;

            while (STOP == FALSE && alarmEnabled == TRUE) {

                read(fd, UA_buffer, 1);

                switch(UA_buffer[0]) {
                    case A_UA:
                        if (state == FLAG)
                            state = A_UA;
                        else
                            state = START;
                        break;

                    case C_UA:
                        if (state == A_UA)
                            state = C_UA;
                        else
                            state = START;
                        break;

                    case (BCC1_UA):
                        if (state == C_UA)
                            state = BCC1_UA;
                        else
                            state = START;
                        break;

                    case FLAG:
                        if (state == BCC1_UA) {
                            STOP = TRUE;
                            state = START;
                            result = 1;

                            alarm(0);   // alarm is disabled
                        }
                        else
                            state = FLAG;
                        break;
                    default:
                        state = START;
                }
            }
        }
    }
    return result;
}

//
--------------------------------------------------------------------------
--------------
//
--------------------------------------------------------------------------
--------------
//
--------------------------------------------------------------------------
--------------

int linkRx(LinkLayer connection) {

    unsigned char buf;
    unsigned char state = START;
    int result = -1;

    while (STOP == FALSE)
    {
        // Returns after 1 char has been input
        read(fd, &buf, 1);

        switch (buf) {
            case 0x03:
              if (state == FLAG) {
                state = A_SET;
              }
              else if (state == A_SET) {
                state = C_SET;
              }
              else {
                state = START;
              }
              break;

            case (BCC1_SET):
              if (state == C_SET) {
                state = BCC1_SET;
              }
              else {
                state = START;
              }
              break;

            case FLAG:
              if (state == BCC1_SET) {
                STOP = TRUE;
                state = START;
                result = 1;

                // Sending the response (UA)
                sendSupervisionFrame(A_UA, C_UA);
                sleep(1);
              }
              else {
                state = FLAG;
              }
              break;
            default:
                state = START;
            }
        }

    return result;
}

// Alarm function handler
void alarmHandler(int signal)
{
    alarmEnabled = FALSE;
    alarmCount++;

    printf("\nAlarm #%d\nStarting retransmission..\n\n", alarmCount);
}

int sendSupervisionFrame(unsigned char A, unsigned char C) {
    unsigned char buf[5] = {FLAG, A, C, A ^ C, FLAG};
    return write(fd, buf, 5);
}

int makeConnection(const char* serialPort) {
    // Open serial port device for reading and writing, and not as
controlling tty
    // because we don't want to get killed if linenoise sends CTRL-C.

    fd = open(serialPort, O_RDWR | O_NOCTTY);

    if (fd < 0)
    {
        perror(serialPort);
        exit(-1);
    }

    struct termios oldtio;
    struct termios newtio;

    // Save current port settings
    if (tcgetattr(fd, &oldtio) == -1)
    {
        perror("tcgetattr");
        exit(-1);
    }

    // Clear struct for new port settings
    memset(&newtio, 0, sizeof(newtio));

    newtio.c_cflag = BAUDRATE | 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; // Inter-character timer unused
    newtio.c_cc[VMIN] = 0;  // Blocking read until 1 char received

    // VTIME e VMIN should be changed in order to protect with a
    // timeout the reception of the following character(s)

    // Now clean the line and activate the settings for the port
    // tcflush() discards data written to the object referred to
    // by fd but not transmitted, or data received but not read,
    // depending on the value of queue_selector:
    //   TCIFLUSH - flushes data received but not read.
    tcflush(fd, TCIOFLUSH);

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

    //printf("New termios structure set\n");

    return fd;
}

unsigned char* getControlPacket(const unsigned int c, const char*
filename, long int length, unsigned int* size) {
    const int L1 = (int) ceil(log2f((float)length)/8.0);
    const int L2 = strlen(filename);
    *size = 1 + 2 + L1 + 2 + L2;
    unsigned char *controlPacket = (unsigned char*)malloc(*size);

    unsigned int i = 0;
    controlPacket[i++] = c;       // control field: 2 - start, 3 - end
    controlPacket[i++] = 0;       // T: 0 - file size
    controlPacket[i++] = L1;      // L

    for (unsigned char j = 0 ; j < L1 ; j++) {      // V
        controlPacket[2 + L1 - j] = length & 0xFF;
        length >>= 8;
    }
    i += L1;
    controlPacket[i++] = 1;
    controlPacket[i++] = L2;
    memcpy(controlPacket + i, filename, L2);

    return controlPacket;
}

unsigned char* getDataPacket(unsigned char sequence, unsigned char *data,
int dataSize, int *packetSize) {
    *packetSize = 1 + 1 + 2 + dataSize;
    unsigned char* packet = (unsigned char*)malloc(*packetSize);

    packet[0] = 1;
    packet[1] = sequence;
    packet[2] = dataSize >> 8 & 0xFF;
    packet[3] = dataSize & 0xFF;
    memcpy(packet+4, data, dataSize);

    return packet;
}

unsigned char* parseControlPacket(unsigned char* packet, int size,
unsigned long int *fileSize) {
    // File Size
    unsigned char fileSizeNBytes = packet[2];
    unsigned char fileSizeAux[fileSizeNBytes];
    memcpy(fileSizeAux, packet + 3, fileSizeNBytes);
    for (unsigned int i = 0; i < fileSizeNBytes; i++)
        *fileSize |= (fileSizeAux[fileSizeNBytes - i - 1] << (8 * i));

    // File Name
    unsigned char fileNameNBytes = packet[3 + fileSizeNBytes + 1];
    unsigned char *name = (unsigned char*)malloc(fileNameNBytes);
    memcpy(name, packet + 3 + fileSizeNBytes + 2, fileNameNBytes);
    return name;
}

void parseDataPacket(const unsigned char* packet, const unsigned int
packetSize, unsigned char* buffer) {
    memcpy(buffer, packet + 4, packetSize - 4);
    buffer += packetSize + 4;
}

unsigned char * getData(FILE* fd, long int fileLength) {
    unsigned char* content = (unsigned char*)malloc(sizeof(unsigned char)
* fileLength);
    fread(content, sizeof(unsigned char), fileLength, fd);
    return content;
}

unsigned char readControlFrame() {

    unsigned char byte, cField = 0;
    LinkLayerState state = START_TX;

    while (state != STOP_R && alarmEnabled == TRUE) {
        int bytes = read(fd, &byte, 1);
        if (bytes > 0) {
            switch (state) {
                case START_TX:
                    if (byte == FLAG) state = FLAG_RCV;
                    break;
                case FLAG_RCV:
                    if (byte == A_RE) state = A_RCV;
                    else if (byte != FLAG) state = START_TX;
                    break;
                case A_RCV:
                    if (byte == C_RR(0) || byte == C_RR(1) || byte ==
C_REJ(0) || byte == C_REJ(1) || byte == C_DISC){
                        state = C_RCV;
                        cField = byte;
                    }
                    else if (byte == FLAG) state = FLAG_RCV;
                    else state = START_TX;
                    break;
                case C_RCV:
                    if (byte == (A_RE ^ cField)) state = BCC1_OK;
                    else if (byte == FLAG) state = FLAG_RCV;
                    else state = START_TX;
                    break;
                case BCC1_OK:
                    if (byte == FLAG) {
                        state = STOP_R;
                    }
                    else state = START_TX;
                    break;
                default:
                    break;
            }
        }
    }
    return cField;
}
```