Faculdade de Engenharia da Universidade do Porto

LAB 1 - DATA LINK

Students and Authors:

Afonso Domingues up202207313@fe.up.pt

Duarte Assunção up202208319@fe.up.pt

Introduction

This report details the process and results of implementing a system for reliable image transfer over a serial port. The primary goal of the project was to design and develop a data transfer protocol capable of ensuring the accurate and complete transmission of the image file penguin.gif between two devices, even in the presence of transmission errors.

The report is organized into sections addressing different aspects of the project, including the architecture and code structure of the implementation, specific use cases and the sequence of function calls, an explanation of the logical and application-level protocols, validation results, and an analysis of data link efficiency. Each section provides a detailed description, supplemented with code excerpts and quantitative data where applicable.

Architecture

The system architecture is structured into two main layers: the Application Layer and the Link Layer, each responsible for distinct tasks to ensure reliable communication over a serial port.

- **Application Layer**: Manages high-level tasks such as file input/output packet construction, and overall control of transmission and reception. It initiates data transfer and relies on the Link Layer for error detection and correction.
- **Link Layer**: Serves as the intermediary between the Application Layer and Physical Layer of communication, managing the connection state, transmitting and receiving data frames, and ensuring data integrity with checksums or acknowledgments. It also handles error detection, retransmissions, and timeouts for robust data exchange.

Functional Blocks and Interfaces
Both layers include functional blocks that work through defined interfaces to maintain efficient and reliable data transfer.

- **Application Layer Blocks**:
    - **Control Packet Handling:** Constructs packets containing essential metadata (e.g., file size, name) to initiate and conclude the transfer.
    - **Data Packet Management:** Breaks the image file into data packets suitable for transmission.
    - **File Management:** Manages file reading and writing operations throughout the transfer process.
- **Link Layer Blocks**:
    - **Error Detection and Correction:** Uses checksums and acknowledgments to detect transmission errors and initiate retransmission for lost packets.
    - **Frame Transmission Management:** Manages the transmission of frames between sender and receiver, coordinating acknowledgment frames to verify successful reception and prompt any necessary retransmissions.

- **Retransmission Logic:** Sets and monitors timeouts to determine when to retransmit frames, ensuring data reliability in error-prone environments.

Interfaces
The Application and Link Layers interact through interfaces that streamline data exchange and control flow. Control packets from the Application Layer are processed by the Link Layer to start and stop data transmission. The Link Layer transmits frames between the sender and receiver, providing acknowledgment and error feedback to the Application Layer, which adjusts transmission as needed to ensure data is accurately and completely transferred.

Code Structure

The code is organized into modular components that enable reliable image transfer over a serial port, with each module managing specific tasks. The header files, application_layer.h and link_layer.h, define the interfaces for their respective layers, including function declarations and data structures for communication management.

The main application logic is encapsulated in the applicationLayer function, which controls the overall workflow for image transmission and reception, utilizing the link layer to handle data transfer. Core functions in the link layer include **llopen**, which opens and initializes the connection; **llwrite**, responsible for sending data packets with retry logic; **llread**, which receives packets and performs error checks; and **llclose**, which closes the connection and frees resources.

Key data structures support communication and error management. The **LinkLayer** structure stores communication parameters such as serialPort for the name of the serial port, role to specify whether the layer is a transmitter or receiver, baudRate, nRetransmissions, and timeout for transmission settings. The **Packet** structure defines the data packets exchanged, including the packet type (control or data), a sequence number for identification, the size of the data being transmitted, and an array to hold the actual data.

The architecture is designed around the interaction between the application layer and the link layer. The application layer uses the **LinkLayer** structure to configure communication settings, constructs packets with the **Packet** structure, and employs **llwrite** for sending these packets while managing retransmissions as needed. The link layer's **llread** function processes incoming data packets, ensuring they are correctly interpreted and buffered for the application layer. Error detection and correction mechanisms are integrated into **llwrite** and **llread**, aligning with the system's overall goal of ensuring reliable data transmission.

Main Use Cases

The system manages reliable image transfer over a serial port through **four main use cases: initializing the connection**, **sending an image**, **receiving an image**, and **closing the connection**. Each use case highlights key function sequences.

1. **Establishing a Connection**
    - Objective: Open the serial port and establish a connection.
    - Sequence:
        - **llopen():** Initializes the link layer and opens the port, sending a SET frame to initiate. Awaits a UA frame from the remote device to confirm the link is ready.
2. **Sending an Image**
    - Objective: Transmit the image data from sender to receiver.
    - Sequence:
        - **applicationLayer()**: Starts transmission, loading the image file into memory.
        - **llwrite()**: Sends each data packet with retries if needed. The application layer monitors for any retransmissions, ensuring the full image is successfully sent.
3. **Receiving an Image**
    - Objective: Receive and buffer the image data.
    - Sequence:
        - **llread():** Receives each data packet, sends acknowledgments (ACKs) for successful receptions, and requests retransmissions if errors occur. Buffers received packets to reconstruct the full image.
4. **Closing the Connection**
    - Objective: Terminate the session and free resources.
    - Sequence:
        - **llclose():** Sends a DISC frame, awaits DISC acknowledgment from the receiver, confirms with a UA frame, and releases allocated resources.

Logical Link Protocol

The Logical Link Protocol (LLP) is designed to facilitate reliable data transfer over the serial link by managing packet transmission and ensuring error handling. The primary functions include establishing connections, sending and receiving packets, and managing retransmissions in case of errors.

Implementation Strategy:

Connection Establishment: The **llopen()** function initiates the link layer, preparing it for data transmission. It establishes a connection by exchanging SET and UA frames.

1. Configures the serial port for usage with function `openSerialPort`
2. Stores the important values for the whole transmission - `role`, `timeout` and `nRetransmissions` - in global variables.

Transmitter

3. Writes the SET frame to the serial port:
4. Waits to receive an UA frame:

```
C/C++
writeBytesSerialPort(set_fr
ame, ASW_BUF_SIZE);
//State Machine to properly
read the UA_frame
//And all the
retransmission logic in
case of error
```

Receiver

3. Waits to receive a SET frame
4. Writes the UA frame to the serial port:

```
C/C++
//State Machine to properly
read the SET frame
writeBytesSerialPort(ua_fra
me, ASW_BUF_SIZE);
```

```
C/C++

unsigned char set_frame[ASW_BUF_SIZE] = {FLAG, ADDRESS_SNDR,
CONTROL_SET, ADDRESS_SNDR ^ CONTROL_SET, FLAG};

unsigned char ua_frame[ASW_BUF_SIZE] = {FLAG, ADDRESS_SNDR,
CONTROL_UA, ADDRESS_SNDR ^ CONTROL_UA, FLAG};
```

Data Transmission: The **llwrite()** function is responsible for sending data packets. It incorporates a mechanism to handle retransmissions based on defined timeouts and errors in the frames.

1. Allocates a buffer `stuffed_buf` to store the stuffed data that will be sent through the serial port. Writes in that buffer as it reads from the buffer passed by the application layer.
2. Writes the whole frame to the serial port:

```
C/C++
//Stuffing Mechanism
writeBytesSerialPort(stuffed_buf, j);
//State Machine to read ACK and whole retransmission mechanism
```

3. Waits for a response:
    a. If it receives a Receiver Ready frame, returns successfully the number of bytes in the frame;

b. If it receives a Rejected frame, retransmits the frame and awaits again for a response;
c. If after a certain amount of time a response is not received a timeout is called and the frame is retransmitted;
d. The retransmission mechanism is performed a given number of times.

Data Receiving: The **llread()** function retrieves incoming packets from the transmitter. It checks for errors and confirms the integrity of received data.

1. Receives a frame and uses the state machine in **Fig. 1** to interpret it.
2. Sends response:
    a. If receives a SET frame, sends the UA frame.
    b. If receives a frame with no errors, replies with a Receiver Ready frame.
    c. If receives a frame with an error in the data bytes (BCC2), returns a Rejected frame.
    d. If receives a frame with an error in first the checksum (BCC1), it goes back to the starting state, reads all the remaining bytes of the frame and waits for a timeout to occur.

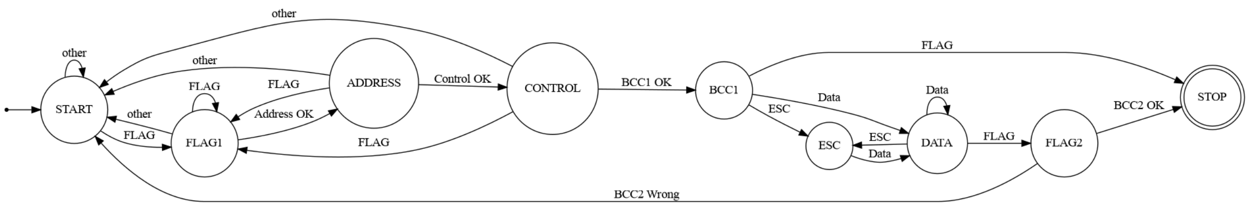


Fig. 1 - State machine for llread()

Connection Termination: The **llclose()** function terminates the session by sending a DISC frame, waiting for a DISC acknowledgment from the receiver, confirming with a UA frame, and releasing allocated resources.

Transmitter

1. Writes a DISC frame to the serial port:

```C/C++
writeBytesSerialPort(disc_frame_sndr, ASW_BUF_SIZE);
```

2. Waits to receive a DISC frame.
3. Writes a UA frame to the serial port:

```C/C++
//State Machine to properly
```

Receiver

1. Waits to receive a DISC frame.
2. Writes a DISC frame to the serial port:

```C/C++
//State Machine to properly read the DISC frame
writeBytesSerialPort(disc_frame_rcvr, ASW_BUF_SIZE);
//State Machine to properly read the UA frame
```

3. Waits to receive a UA frame.

```
read the DISC frame
writeBytesSerialPort(ua_fra
me_disc, ASW_BUF_SIZE);
```

4. Closes the serial port.

4. Closes the serial port.

```
C/C++

unsigned char disc_frame_sndr[ASW_BUF_SIZE] = {FLAG,
ADDRESS_SNDR, CONTROL_DISC, ADDRESS_SNDR ^ CONTROL_DISC,
FLAG};

unsigned char disc_frame_rcvr[ASW_BUF_SIZE] = {FLAG,
ADDRESS_RCVR, CONTROL_DISC, ADDRESS_RCVR ^ CONTROL_DISC,
FLAG};

unsigned char ua_frame_disc[ASW_BUF_SIZE] = {FLAG,
ADDRESS_RCVR, CONTROL_UA, ADDRESS_RCVR ^ CONTROL_UA, FLAG};
```

Application Protocol

The Application Protocol is responsible for the higher-level functions of the system, including file management, control packet handling, and data packet management. It facilitates the overall workflow of sending and receiving files, ensuring the accurate transmission of data.

Implementation Strategy:

File Management: The application layer handles file operations, ensuring that files are correctly opened, read, written, and closed during the transfer.

```
C/C++
FILE *file = fopen(filename, "r");

if (file == NULL) {

      printf("ERROR: Could not open the file\n");

      exit(1);

}
```

Control Packet Handling: The application layer constructs control packets to encapsulate metadata about the file, such as its size and name, enabling the connection to be initialized or terminated.

C/C++
```c
unsigned char ctrl_packet[size_ctrl_packet];

int i = 0;

ctrl_packet[i++] = PACKET_CTRL_START;

ctrl_packet[i++] = PACKET_TYPE_SIZE; // type file size

ctrl_packet[i++] = L1;

for (int v = L1 - 1; v >= 0; v--) ctrl_packet[i++] =
(file_size >> (8 * v) & 0xFF);

ctrl_packet[i++] = PACKET_TYPE_NAME; // type filename

ctrl_packet[i++] = L2;

memcpy(&ctrl_packet[i], filename, L2);

i += L2;

llwrite(ctrl_packet, size_ctrl_packet);
```

Data Packet Management: Data packets are created and sent in manageable sizes, in this case with a size of 1024 bytes. The application layer segments the file content and communicates with the link layer for reliable transmission.

C/C++
```c
unsigned char data_packet[PACKET_SIZE];

//Construction of Data Packets

llwrite(data_packet, size_data_packet);
```

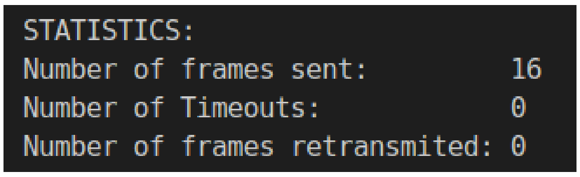
Receiving Data: The application layer also manages the receipt of data packets, from the link layer reading them into a buffer and ensuring that data is written correctly to the intended file.

```c
C/C++
While (TRUE){
int read_packet_size = llread(packet);

if (packet[0] == PACKET_CTRL_END) break; //Received the
Control Packet to end connection

else fwrite(packet + 4, 1, read_packet_size - 4, new_file); //
Writes data to file

}
```

Validation

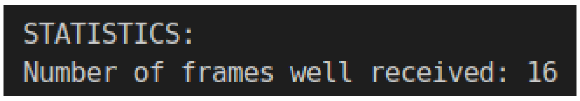
When running normally, the code executes without errors, timeouts, or retransmissions, as expected:

Transmitter



```
STATISTICS:
Number of frames sent:         16
Number of Timeouts:            0
Number of frames retransmited: 0
```

Receiver

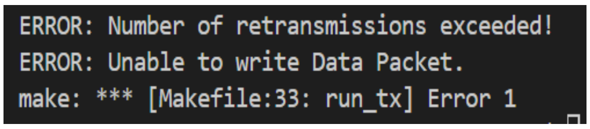


```
STATISTICS:
Number of frames well received: 16
```

When introducing errors, the code executes well for any ber <= 0.00006. Otherwise, the transmission fails:

Transmitter

ber = 0.00007



```
ERROR: Number of retransmissions exceeded!
ERROR: Unable to write Data Packet.
make: *** [Makefile:33: run_tx] Error 1
```

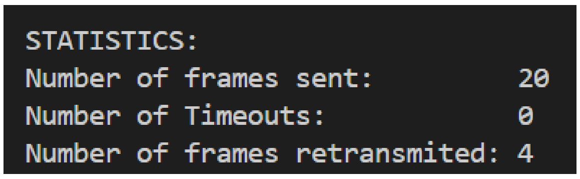
Receiver

ber = 0.00007

Receiver does not receive any frame after the establishment of connection.

Transmitter

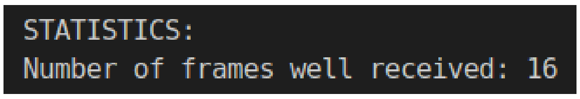
ber = 0.00006



```
STATISTICS:
Number of frames sent:         20
Number of Timeouts:            0
Number of frames retransmited: 4
```

Receiver

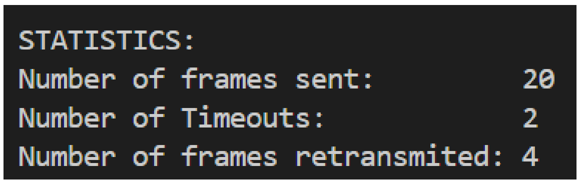
ber = 0.00006



```
STATISTICS:
Number of frames well received: 16
```

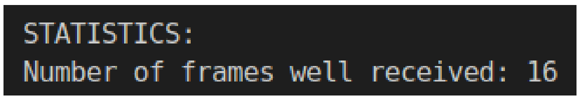
When disconnecting the cable in the middle of the transmission, it performs well:

Transmitter



```
STATISTICS:
Number of frames sent:          20
Number of Timeouts:             2
Number of frames retransmited: 4
```

Receiver



```
STATISTICS:
Number of frames well received: 16
```

Data Link Protocol Efficiency

Comparison of theoretical and experimental efficiency:

Baud Rate	9600	19200	38400	4800
Propagation Time	0μs	1041μs	9895μs	4166μs
Theoretical Transmission Time	0.106667s	0.053333s	0.026667s	0.213333s
Theoretical Total Time	0.106667s	0.055415s	0.046457s	0.221665s
Theoretical Efficiency	1	0.9624	0.5740	0.9624
Experimental Transmission Time	1.078963s	0.538788s	0.279804s	2.159821s
Experimental Total Time	1.084280s	0.544182s	0.290773s	2.174312s
Experimental Efficiency	0.9951	0.9901	0.9623	0.9933

Conclusion

This report presented the design, implementation, and evaluation of a system for reliable image transfer over a serial port, using a Stop & Wait-based data link protocol. By structuring the system into distinct application and link layers, we established a clear protocol architecture that ensured both reliable data transmission and modular organization. Key components, including control packet handling, data packet management, and error handling, were designed to ensure accurate and complete image transmission. Efficiency testing highlighted the protocol's reliability and underscored the inherent trade-offs in throughput associated with Stop & Wait.

Appendix I - Source code

state_machine.h

```
C/C++
#ifndef _STATE_MACHINE_H_
#define _STATE_MACHINE_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 <stdbool.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 1024

#define FLAG 0x7E
#define ADDRESS_SNDR 0x03
#define ADDRESS_RCVR 0x01

#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
#define CONTROL_B0 0X00
#define CONTROL_B1 0x80
#define ESC 0X7D


typedef enum {
       START_STATE,
       FLAG_STATE,
       ADDRESS_STATE,
       CONTROL_STATE,
       BCC1_STATE,
       DATA_STATE,
       ESC_STATE,
       FLAG2_STATE,
       STOP_STATE
} frameState_t;

#endif // _STATE_MACHINE_H_
```

application_layer.c

```
C/C++
// Application layer protocol implementation

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

#define PACKET_SIZE 1024

#define PACKET_CTRL_START 1
#define PACKET_CTRL_DATA 2
#define PACKET_CTRL_END 3

#define PACKET_TYPE_SIZE 0
#define PACKET_TYPE_NAME 1

void applicationLayer(const char *serialPort, const char *role, int
baudRate,
                      int nTries, int timeout, const char *filename)
{
        LinkLayer linklayer;
        strcpy(linklayer.serialPort, serialPort);
        if (!strcmp(role, "rx"))
        linklayer.role = LlRx;
        else
        linklayer.role = LlTx;
        linklayer.baudRate = baudRate;
        linklayer.nRetransmissions = nTries;
        linklayer.timeout = timeout;

        unsigned char packet[PACKET_SIZE * 2];
        if (llopen(linklayer) != 1)
        {
        printf("ERROR: Unable to open serial port\n");
        exit(1);
        }

        switch (linklayer.role)
        {
        case LlTx:
        {
        FILE *file = fopen(filename, "r");
        if (file == NULL)
        {
                printf("ERROR: Could not open the file\n");
                exit(1);
        }

        fseek(file, 0, SEEK_END);        // redirects the pointer to the end
of the file
        long int file_size = ftell(file); // returns the current offset from
the starting position
        fseek(file, 0, SEEK_SET);        // redirects the pointer back to the
start of the file

        int L1 = 8; // ceil(log2((double)file_size) / 8.0);
        unsigned int L2 = strlen(filename);
        if (L2 > 255)
        {
                printf("ERROR: File name too big!\n");
                exit(1);
        }
        unsigned int size_ctrl_packet = 1 + 2 + L1 + 2 + L2; // {CTRL, T1, L1,
V1[L1], T2, L2, V2[L2]}

        unsigned char ctrl_packet[size_ctrl_packet];
        int i = 0;
        ctrl_packet[i++] = PACKET_CTRL_START;
        ctrl_packet[i++] = PACKET_TYPE_SIZE; // type file size
        ctrl_packet[i++] = L1;
        for (int v = L1 - 1; v >= 0; v--)
        {
                ctrl_packet[i++] = (file_size >> (8 * v) & 0xFF);
        }
        ctrl_packet[i++] = PACKET_TYPE_NAME; // type filename
        ctrl_packet[i++] = L2;
        memcpy(&ctrl_packet[i], filename, L2);
        i += L2;
```

```
if (llwrite(ctrl_packet, size_ctrl_packet) == -1)
{
        printf("ERROR: Control packet not written properly!\n");
        fclose(file);
        exit(1);
}

int sequence_number = 0;
unsigned char data_packet[PACKET_SIZE];

int bytesToSend = file_size;
unsigned char *file_content = (unsigned char *)calloc(file_size, 1);
fread(file_content, 1, file_size, file);
unsigned long offset = 0;

while (bytesToSend > 0)
{
        printf("INFO: Bytes Still to send:%d\n", bytesToSend);
        i = 0;
        int data_size;
        if (bytesToSend < PACKET_SIZE - 4)
        data_size = bytesToSend;
        else
        data_size = PACKET_SIZE - 4;

        bytesToSend -= data_size;

        int size_data_packet = 4 + data_size;
        data_packet[i++] = PACKET_CTRL_DATA;
        data_packet[i++] = sequence_number++ % 100;
        data_packet[i++] = (data_size >> 8) & 0xFF;
        data_packet[i++] = data_size & 0xFF;
        memcpy(data_packet + 4, file_content + offset, data_size);
        offset += data_size;

        if (llwrite(data_packet, size_data_packet) == -1)
        {
        printf("ERROR: Unable to write Data Packet.\n");
        fclose(file);
        exit(1);
        }
}
free(file_content);
ctrl_packet[0] = PACKET_CTRL_END;
if (llwrite(ctrl_packet, size_ctrl_packet) == -1)
{
        printf("ERROR: Final Control packet not written properly!\n");
        fclose(file);
```

```
                exit(1);
        }

        fclose(file);
        break;
        }

        case LlRx:
        {
        int ctrl_size = llread(packet);
        if (ctrl_size == -1)
        {
                printf("ERROR: Unable to read Control Packet from link
layer.\n");
                exit(1);
        }
        printf("INFO: Nº de bytes lidos:%d\n", ctrl_size);
        unsigned long fileSize = 0;

        for (unsigned i = 0; i < 8; i++)
        {
                fileSize += packet[3 + i] << 8 * (8 - i - 1);
        }

        FILE *new_file = fopen(filename, "w");

        while (TRUE)
        {
                int read_packet_size = 0;
                read_packet_size = llread(packet);
                if (read_packet_size == -1)
                {
                printf("ERROR: Unable to read Data Packet from link layer.\n");
                fclose(new_file);
                exit(1);
                }

                if (packet[0] == PACKET_CTRL_END)
                break;
                else
                {
                fwrite(packet + 4, 1, read_packet_size - 4, new_file);
                }
        }
        fclose(new_file);
        break;
        }
        default:
```

```
        break;
        }

        if (llclose(TRUE) != 1)
        {
        printf("ERROR: Unable to close serial port\n");
        exit(1);
        }
        printf("INFO: Serial port successfully closed!\n");

        return;
}
```

link_layer.c

```
C/C++
// Link layer protocol implementation

#include "link_layer.h"
#include "serial_port.h"
#include "state_machine.h"

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

#define BUF_SIZE 1024
#define ASW_BUF_SIZE 5

#define FLAG 0x7E
#define ADDRESS_SNDR 0x03
#define ADDRESS_RCVR 0x01

#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
#define CONTROL_B0 0X00
#define CONTROL_B1 0x80

unsigned char rr_0[ASW_BUF_SIZE] = {FLAG, ADDRESS_SNDR, CONTROL_RR0,
ADDRESS_SNDR ^ CONTROL_RR0, FLAG};
unsigned char rr_1[ASW_BUF_SIZE] = {FLAG, ADDRESS_SNDR, CONTROL_RR1,
ADDRESS_SNDR ^ CONTROL_RR1, FLAG};
```

```
unsigned char rej_0[ASW_BUF_SIZE] = {FLAG, ADDRESS_SNDR, CONTROL_REJ0,
ADDRESS_SNDR ^ CONTROL_REJ0, FLAG};
unsigned char rej_1[ASW_BUF_SIZE] = {FLAG, ADDRESS_SNDR, CONTROL_REJ1,
ADDRESS_SNDR ^ CONTROL_REJ1, FLAG};
unsigned char disc_frame_sndr[ASW_BUF_SIZE] = {FLAG, ADDRESS_SNDR,
CONTROL_DISC, ADDRESS_SNDR ^ CONTROL_DISC, FLAG};
unsigned char disc_frame_rcvr[ASW_BUF_SIZE] = {FLAG, ADDRESS_RCVR,
CONTROL_DISC, ADDRESS_RCVR ^ CONTROL_DISC, FLAG};
unsigned char ua_frame[ASW_BUF_SIZE] = {FLAG, ADDRESS_SNDR, CONTROL_UA,
ADDRESS_SNDR ^ CONTROL_UA, FLAG};
unsigned char ua_frame_disc[ASW_BUF_SIZE] = {FLAG, ADDRESS_RCVR, CONTROL_UA,
ADDRESS_RCVR ^ CONTROL_UA, FLAG};
unsigned char set_frame[ASW_BUF_SIZE] = {FLAG, ADDRESS_SNDR, CONTROL_SET,
ADDRESS_SNDR ^ CONTROL_SET, FLAG};

volatile int STOP = FALSE;

int alarmEnabled = FALSE;

int alarmCount;
LinkLayerRole role;
int timeout = 0;
int nRetransmissions = 0;
int nTries = 0;

int frame_n = 0;

typedef struct
{
	unsigned int n_frames;
	unsigned int n_timeouts;
	int n_retransmissions;

} Statistics;

Statistics statistics = {0};

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

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

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

       role = connectionParameters.role;
       timeout = connectionParameters.timeout;
       nRetransmissions = connectionParameters.nRetransmissions;

       frameState_t state;
       switch (connectionParameters.role)
       {
       case LlTx:
       (void)signal(SIGALRM, alarmHandler);

       alarmCount = 0;

       statistics.n_retransmissions--;
       while (alarmCount < nRetransmissions)
       {
              if (alarmEnabled == FALSE)
              {
              alarm(timeout); // Set alarm to be triggered in timeout seconds
              alarmEnabled = TRUE;

              int bytes = writeBytesSerialPort(set_frame, ASW_BUF_SIZE);
              printf("INFO: %d bytes written\n", bytes);
              statistics.n_frames++;
              statistics.n_retransmissions++;

              state = START_STATE;
              while (alarmEnabled != FALSE && state != STOP_STATE)
              {
                     unsigned char byte_read = 0;
                     int n_bytes_read = readByteSerialPort(&byte_read);
                     if (n_bytes_read == 0)
                            continue;

                     switch (state)
                     {
                     case START_STATE:
                            if (byte_read == FLAG)
                            {
                            state = FLAG_STATE;
                            // printf("First flag received\n");
```

```
        }
        break;

    case FLAG_STATE:
        if (byte_read == ADDRESS_SNDR)
        {
        state = ADDRESS_STATE;
        // printf("Adress received\n");
        }
        else if (byte_read != FLAG)
        {
        state = START_STATE;
        // printf("Back to start :(\n");
        }
        break;

    case ADDRESS_STATE:
        if (byte_read == CONTROL_UA)
        {
        state = CONTROL_STATE;
        // printf("Control received\n");
        }
        else if (byte_read == FLAG)
        {
        state = FLAG_STATE;
        // printf("Flag received instead of control\n");
        }
        else
        {
        state = START_STATE;
        // printf("Back to start from address\n");
        }
        break;

    case CONTROL_STATE:
        if (byte_read == (ADDRESS_SNDR ^ CONTROL_UA))
        {
        state = BCC1_STATE;
        // printf("BCC received\n");
        }
        else if (byte_read == FLAG)
        {
        state = FLAG_STATE;
        // printf("Flag received instead of BCC\n");
        }
        else
        {
        state = START_STATE;
```

```
                // printf("Back to start from control\n");
                }
                break;

            case BCC1_STATE:
                if (byte_read == FLAG)
                {
                state = STOP_STATE;
                alarm(0);
                alarmEnabled = FALSE;
                printf("INFO: Connection established
successfuly\n");
                frame_n = 0;
                return 1;
                }
                else
                {
                state = START_STATE;
                // printf("All the way to the start from BCC\n");
                }
                break;

            default:
                break;
            }
        }
        }
    }
    printf("TIMEOUT: Could not establish connection\n");
    break;

    case LlRx:
    state = START_STATE;
    while (state != STOP_STATE)
    {
        unsigned char byte_read = 0;
        int n_bytes_read = readByteSerialPort(&byte_read);
        if (n_bytes_read == 0)
        continue;
        switch (state)
        {
        case START_STATE:
        if (byte_read == FLAG)
        {
            state = FLAG_STATE;
            // printf("First flag received\n");
        }
        break;
```

```
case FLAG_STATE:
if (byte_read == ADDRESS_SNDR)
{
        state = ADDRESS_STATE;
        // printf("Adress received\n");
}
else if (byte_read != FLAG)
{
        state = START_STATE;
        // printf("Back to start :(\n");
}
break;

case ADDRESS_STATE:
if (byte_read == CONTROL_SET)
{
        state = CONTROL_STATE;
        // printf("Control received\n");
}
else if (byte_read == FLAG)
{
        state = FLAG_STATE;
        // printf("Flag received instead of control\n");
}
else
{
        state = START_STATE;
        // printf("Back to start from address\n");
}
break;

case CONTROL_STATE:
if (byte_read == (ADDRESS_SNDR ^ CONTROL_SET))
{
        state = BCC1_STATE;
        // printf("BCC received\n");
}
else if (byte_read == FLAG)
{
        state = FLAG_STATE;
        // printf("Flag received instead of BCC\n");
}
else
{
        state = START_STATE;
        // printf("Back to start from control\n");
}
```

```
                break;

                case BCC1_STATE:
                if (byte_read == FLAG)
                {
                        state = STOP_STATE;
                        // printf("Last flag received proceded to stop\n");
                }
                else
                {
                        state = START_STATE;
                        // printf("All the way to the start from BCC\n");
                }
                break;

                default:
                break;
                }
        }
        printf("INFO: Connection stablished successfully!\n");
        int bytes = writeBytesSerialPort(ua_frame, ASW_BUF_SIZE);
        if (bytes != 5)
                printf("ERROR: Failed to send 5 bytes (UA frame).\n");
        frame_n = 0;
        statistics.n_frames++;
        return 1;
        break;

        default:
        break;
        }

        return -1;
}

////////////////////////////////////////////////
// LLWRITE
////////////////////////////////////////////////
int llwrite(const unsigned char *buf, int bufSize)
{
        // buf 0 S L1 L2
        //
        alarmCount = 0;
        bool is_rej = false;

        // BYTE STUFFING:
        unsigned char stuffed_buf[BUF_SIZE * 2] = {0};
        unsigned j = 0;
```

```
stuffed_buf[j++] = FLAG;
stuffed_buf[j++] = ADDRESS_SNDR;
if (frame_n == 0)
{
stuffed_buf[j++] = CONTROL_B0;
stuffed_buf[j++] = CONTROL_B0 ^ ADDRESS_SNDR;
}
else
{
stuffed_buf[j++] = CONTROL_B1;
stuffed_buf[j++] = CONTROL_B1 ^ ADDRESS_SNDR;
}

unsigned int bcc2 = buf[0];

for (int i = 1; i < bufSize; i++)
bcc2 ^= buf[i];

for (int i = 0; i < bufSize; i++)
{
if (buf[i] == FLAG || buf[i] == ESC)
{
       stuffed_buf[j++] = ESC;
       stuffed_buf[j++] = buf[i] ^ 0x20;
}
else
       stuffed_buf[j++] = buf[i];
}
stuffed_buf[j++] = bcc2;
stuffed_buf[j++] = FLAG;

statistics.n_retransmissions--;
while (alarmCount < nRetransmissions)
{
if (alarmEnabled == FALSE)
{
       alarm(timeout); // Set alarm to be triggered in 3s
       alarmEnabled = TRUE;
       is_rej = false;

       int bytes = writeBytesSerialPort(stuffed_buf, j);
       printf("INFO: %d bytes written\n", bytes);
       statistics.n_frames++;
       statistics.n_retransmissions++;

       frameState_t state = START_STATE;
       while (alarmEnabled != FALSE && state != STOP_STATE)
       {
            unsigned char byte_read = 0;
            int n_bytes_read = readByteSerialPort(&byte_read);
            if (n_bytes_read == 0)
                    continue;

            switch (state)
            {
            case START_STATE:
                    if (byte_read == FLAG)
                    {
                            state = FLAG_STATE;
                            // printf("First flag received\n");
                    }
                    break;

            case FLAG_STATE:
                    if (byte_read == ADDRESS_SNDR)
                    {
                            state = ADDRESS_STATE;
                            // printf("Adress received\n");
                    }
                    else if (byte_read != FLAG)
                    {
                            state = START_STATE;
                            // printf("Back to start :(\n");
                    }
                    break;

            case ADDRESS_STATE:
                    if (byte_read == CONTROL_REJ0 || byte_read ==
CONTROL_REJ1)
                    {
                            state = CONTROL_STATE;
                            // printf("Control REJ received\n");
                    }
                    else if (byte_read == CONTROL_RR0 || byte_read ==
CONTROL_RR1)
                    {
                            state = CONTROL_STATE;
                            // printf("Control RR received\n");
                    }
                    else if (byte_read == CONTROL_UA)
                    {
                            state = CONTROL_STATE;
                    }
                    else if (byte_read == FLAG)
                    {
                            state = FLAG_STATE;
```

```
                // printf("Flag received instead of control\n");
            }
            else
            {
                state = START_STATE;
                // printf("Back to start from address\n");
            }
            break;

        case CONTROL_STATE:

            if (byte_read == (ADDRESS_SNDR ^ CONTROL_REJ0) ||
byte_read == (ADDRESS_SNDR ^ CONTROL_REJ1))
            {
                state = BCC1_STATE;
                is_rej = true;
                // printf("BCC received\n");
            }
            else if (byte_read == (ADDRESS_SNDR ^ CONTROL_RR0) ||
byte_read == (ADDRESS_SNDR ^ CONTROL_RR1))
            {
                state = BCC1_STATE;
            }
            else if (byte_read == (ADDRESS_SNDR ^ CONTROL_UA))
            {
                state = BCC1_STATE;
            }
            else if (byte_read == FLAG)
            {
                state = FLAG_STATE;
                // printf("Flag received instead of BCC\n");
            }
            else
            {
                state = START_STATE;
                // printf("Back to start from control\n");
            }
            break;

        case BCC1_STATE:
            if (byte_read == FLAG)
            {
                state = STOP_STATE;
                // printf("Going to stop\n");
            }
            else
            {
                state = START_STATE;
```

```
                        is_rej = false;
                        // printf("All the way to the start from BCC\n");
                    }
                    break;

                default:
                    break;
                }
                }
                if (is_rej && state == STOP_STATE)
                {
                alarm(0);
                alarmEnabled = FALSE;
                printf("INFO: Frame rejected - retrasmiting\n");
                continue;
                }
                else if (!is_rej && state == STOP_STATE)
                {
                alarm(0);
                alarmEnabled = FALSE;
                printf("INFO: Frame Well transmited\n");
                if (frame_n == 0)
                    frame_n = 1;
                else
                    frame_n = 0;
                if (j < 7)
                    return j;
                return j - 7;
                }
        }
        }
        if (is_rej)
        printf("ERROR: Number of retransmissions exceeded!\n");
        else
        printf("TIMEOUT: Could not send the frame\n");

        return -1;
}

////////////////////////////////////////////////
// LLREAD
////////////////////////////////////////////////
int llread(unsigned char *packet)
{
        int bytes;
        int size;
        memset(packet, 0, BUF_SIZE * 2);
```

```
unsigned char address = 0;
unsigned char control = 0;

frameState_t state = START_STATE;
unsigned i = 0;
while (state != STOP_STATE)
{
if (i >= BUF_SIZE)
       size = -2;

unsigned char byte_read = 0;
int n_bytes_read = readByteSerialPort(&byte_read);
// printf("%d\n",n_bytes_read);
if (n_bytes_read == 0 && state != FLAG2_STATE)
       continue;

unsigned char bcc;
switch (state)
{
case START_STATE:
       // printf("First flag received\n");
       if (byte_read == FLAG)
       {
       state = FLAG_STATE;
       // packet[i] = byte_read;
       // i++;
       }
       break;

case FLAG_STATE:
       if (byte_read == ADDRESS_SNDR)
       {
       state = ADDRESS_STATE;
       // packet[i] = byte_read;
       address = byte_read;
       // i++;
       //  printf("Adress received\n");
       }
       else if (byte_read != FLAG)
       {
       state = START_STATE;
       // i = 0;
       //  printf("Back to start :(\n");
       }
       break;

case ADDRESS_STATE:
```

```
            if ((byte_read == CONTROL_B0) || (byte_read == CONTROL_B1) ||
(byte_read == CONTROL_SET))
            {
            state = CONTROL_STATE;
            // packet[i] = byte_read;
            control = byte_read;
            // i++;
            //  printf("Control received\n");
            }
            else if (byte_read == FLAG)
            {
            state = FLAG_STATE;
            // i = 1;
            //  printf("Flag received instead of control\n");
            }
            else
            {
            state = START_STATE;
            // i = 0;
            //  printf("Back to start from address\n");
            }
            break;

      case CONTROL_STATE:
            if (byte_read == (address ^ control))
            {
            state = BCC1_STATE;
            // printf("BCC received\n");
            }
            else if (byte_read == FLAG)
            {
            state = FLAG_STATE;
            // printf("Flag received instead of BCC\n");
            }
            else
            {
            state = START_STATE;
            // printf("Back to start from control\n");
            }
            break;

      case BCC1_STATE:
            if (byte_read == ESC)
            {
            state = ESC_STATE;
            // printf("ESC read\n");
            }
            else if (byte_read == FLAG)
            {
            state = STOP_STATE;
            size = 5;
            }
            else
            {
            state = DATA_STATE;
            packet[i++] = byte_read;
            // printf("Data Byte BCC\n");
            }
            break;

    case ESC_STATE:
            packet[i++] = (byte_read ^ 0x20); // destuffing
            state = DATA_STATE;
            break;

    case DATA_STATE:
            if (byte_read == FLAG)
            {
            state = FLAG2_STATE;
            // printf("Read Flag going to end\n");
            }
            else if (byte_read == ESC)
            {
            state = ESC_STATE;
            // printf("ESC read\n");
            }
            else
            { // Read normal data
            packet[i++] = byte_read;
            // printf("Read normal data byte\n");
            }
            break;

    case FLAG2_STATE:
            // printf("Estou no estado final\n");
            bcc = packet[0];
            for (unsigned v = 1; v < (i - 1); v++)
            bcc ^= packet[v];

            if (bcc == packet[i - 1])
            {
            state = STOP_STATE;
            size = i - 1;
            // printf("Cheguei ao fim\n");
            }
            else
            {
            i = 0;
            state = STOP_STATE;
            // printf("BCC2 wrong: stopping to reject\n");
            size = -1;
            }
            break;

    default:
            break;
    }

    if (state == STOP_STATE)
    {
            if (size == -2)
            {
            printf("ERROR: Allocated buffer is too small.\n");
            return -1;
            }
            else if (size == 5)
            { // Received a SET send a UA_FRAME
            bytes = writeBytesSerialPort(ua_frame, ASW_BUF_SIZE);
            // printf("Sent UA\n");
            size = 0;
            }
            else if (size == -1)
            { // BCC2 WRONG!
            if (control == CONTROL_B0)
            {
                    bytes = writeBytesSerialPort(rej_0, ASW_BUF_SIZE);
                    frame_n = 0;

                    // printf("Sent rej0\n");
                    state = START_STATE;
            }
            else
            {
                    bytes = writeBytesSerialPort(rej_1, ASW_BUF_SIZE);
                    frame_n = 1;

                    // printf("Sent rej1\n");
                    state = START_STATE;
            }
            }
            else if (control == CONTROL_B0 && frame_n == 0)
            {
            bytes = writeBytesSerialPort(rr_1, ASW_BUF_SIZE);
            frame_n = 1;
```

```c
                // printf("Sent RR1\n");
                }
                else if (control == CONTROL_B1 && frame_n == 1)
                {
                bytes = writeBytesSerialPort(rr_0, ASW_BUF_SIZE);
                frame_n = 0;
                // printf("Sent RR0\n");
                }
                else if (control == CONTROL_B1)
                { // duplicated I1
                bytes = writeBytesSerialPort(rr_0, ASW_BUF_SIZE);
                frame_n = 0;
                // printf("Sent nothing - duplicate discard\n");
                state = START_STATE;
                }
                else if (control == CONTROL_B0)
                { // duplicated I0
                bytes = writeBytesSerialPort(rr_1, ASW_BUF_SIZE);
                frame_n = 1;
                // printf("Sent nothing - duplicate discard\n");
                state = START_STATE;
                }
        }
        }
        printf("INFO: Bytes sent in answer:%d\n", bytes);
        printf("INFO: Read packet size:%d\n", size);
        statistics.n_frames++;
        return size;
}

////////////////////////////////////////////////
// LLCLOSE
////////////////////////////////////////////////
int llclose(int showStatistics)
{
        (void)signal(SIGALRM, alarmHandler);

        alarmCount = 0;

        frameState_t state = START_STATE;
        switch (role)
        {
        case LlRx:
        // Receiving DISC frame:
        state = START_STATE;
        while (state != STOP_STATE)
        {
                unsigned char byte_read = 0;
```

```
int n_bytes_read = readByteSerialPort(&byte_read);
if (n_bytes_read == 0)
continue;
switch (state)
{
case START_STATE:
if (byte_read == FLAG)
{
       state = FLAG_STATE;
       // printf("First flag received\n");
}
break;

case FLAG_STATE:
if (byte_read == ADDRESS_SNDR)
{
       state = ADDRESS_STATE;
       // printf("Adress received\n");
}
else if (byte_read != FLAG)
{
       state = START_STATE;
       // printf("Back to start :(\n");
}
break;

case ADDRESS_STATE:
if (byte_read == CONTROL_DISC)
{
       state = CONTROL_STATE;
       // printf("Control received\n");
}
else if (byte_read == FLAG)
{
       state = FLAG_STATE;
       // printf("Flag received instead of control\n");
}
else
{
       state = START_STATE;
       // printf("Back to start from address\n");
}
break;

case CONTROL_STATE:
if (byte_read == (ADDRESS_SNDR ^ CONTROL_DISC))
{
       state = BCC1_STATE;
```

```
                        // printf("BCC received\n");
                }
                else if (byte_read == FLAG)
                {
                        state = FLAG_STATE;
                        // printf("Flag received instead of BCC\n");
                }
                else
                {
                        state = START_STATE;
                        // printf("Back to start from control\n");
                }
                break;

                case BCC1_STATE:
                if (byte_read == FLAG)
                {
                        state = STOP_STATE;
                        statistics.n_frames++;
                        // printf("Last flag received proceded to stop\n");
                }
                else
                {
                        state = START_STATE;
                        // printf("All the way to the start from BCC\n");
                }
                break;

                default:
                break;
                }
        }

        // Sending DISC frame:
        alarmCount = 0;
        while (alarmCount < nRetransmissions)
        {
                if (alarmEnabled == FALSE)
                {
                alarm(timeout);
                alarmEnabled = TRUE;

                int bytes = writeBytesSerialPort(disc_frame_rcvr,
ASW_BUF_SIZE);
                printf("INFO: %d bytes written - RECEIVER DISC\n", bytes);

                state = START_STATE;
                while (alarmEnabled != FALSE && state != STOP_STATE)
        {
                unsigned char byte_read = 0;
                int n_bytes_read = readByteSerialPort(&byte_read);
                if (n_bytes_read == 0)
                        continue;
                switch (state)
                {
                case START_STATE:
                        if (byte_read == FLAG)
                        {
                        state = FLAG_STATE;
                        // printf("First flag received\n");
                        }
                        break;

                case FLAG_STATE:
                        if (byte_read == ADDRESS_RCVR)
                        {
                        state = ADDRESS_STATE;
                        // printf("Adress received\n");
                        }
                        else if (byte_read != FLAG)
                        {
                        state = START_STATE;
                        // printf("Back to start :(\n\n");
                        }
                        break;

                case ADDRESS_STATE:
                        if (byte_read == CONTROL_UA)
                        {
                        state = CONTROL_STATE;
                        // printf("Control received\n");
                        }
                        else if (byte_read == FLAG)
                        {
                        state = FLAG_STATE;
                        // printf("Flag received instead of control\n");
                        }
                        else
                        {
                        state = START_STATE;
                        // printf("Back to start from address\n");
                        }
                        break;

                case CONTROL_STATE:
                        if (byte_read == (ADDRESS_RCVR ^ CONTROL_UA))
                        {
                        state = BCC1_STATE;
                        // printf("BCC received\n");
                        }
                        else if (byte_read == FLAG)
                        {
                        state = FLAG_STATE;
                        // printf("Flag received instead of BCC\n");
                        }
                        else
                        {
                        state = START_STATE;
                        // printf("Back to start from control\n");
                        }
                        break;

                case BCC1_STATE:
                        if (byte_read == FLAG)
                        {
                        state = STOP_STATE;
                        alarm(0);
                        alarmEnabled = FALSE;
                        printf("INFO: Successfully disconnected\n");
                        statistics.n_frames++;

                        if (showStatistics)
                        {
                                printf("\nSTATISTICS:\n");
                                printf("Number of frames well received:
%d\n", statistics.n_frames);
                        }
                        return 1;
                        }
                        else
                        {
                        state = START_STATE;
                        // printf("All the way to the start from BCC\n");
                        }
                        break;

                default:
                        break;
                }
        }
        }
    }
    printf("TIMEOUT: UA frame not received!\n");
    break;
```

```
        case LlTx:
        alarmCount = 0;
        statistics.n_retransmissions--;
        while (alarmCount < nRetransmissions)
        {
                if (alarmEnabled == FALSE)
                {
                alarm(timeout); // Set alarm to be triggered in 3s
                alarmEnabled = TRUE;

                int bytes = writeBytesSerialPort(disc_frame_sndr,
ASW_BUF_SIZE);
                printf("INFO: %d bytes written - SENDER DISC\n", bytes);
                statistics.n_frames++;
                statistics.n_retransmissions++;

                state = START_STATE;
                while (alarmEnabled != FALSE && state != STOP_STATE)
                {
                        unsigned char byte_read = 0;
                        int n_bytes_read = readByteSerialPort(&byte_read);
                        if (n_bytes_read == 0)
                                continue;

                        switch (state)
                        {
                        case START_STATE:
                                if (byte_read == FLAG)
                                {
                                state = FLAG_STATE;
                                // printf("First flag received\n");
                                }
                                break;

                        case FLAG_STATE:
                                if (byte_read == ADDRESS_RCVR)
                                {
                                state = ADDRESS_STATE;
                                // printf("Adress received\n");
                                }
                                else if (byte_read != FLAG)
                                {
                                state = START_STATE;
                                // printf("Back to start :(\n");
                                }
                                break;
                        case ADDRESS_STATE:
                                if (byte_read == CONTROL_DISC)
                                {
                                state = CONTROL_STATE;
                                // printf("Control received\n");
                                }
                                else if (byte_read == FLAG)
                                {
                                state = FLAG_STATE;
                                // printf("Flag received instead of control\n");
                                }
                                else
                                {
                                state = START_STATE;
                                // printf("Back to start from address\n");
                                }
                                break;

                        case CONTROL_STATE:
                                if (byte_read == (ADDRESS_RCVR ^ CONTROL_DISC))
                                {
                                state = BCC1_STATE;
                                // printf("BCC received\n");
                                }
                                else if (byte_read == FLAG)
                                {
                                state = FLAG_STATE;
                                // printf("Flag received instead of BCC\n");
                                }
                                else
                                {
                                state = START_STATE;
                                // printf("Back to start from control\n");
                                }
                                break;

                        case BCC1_STATE:
                                if (byte_read == FLAG)
                                {
                                state = STOP_STATE;
                                alarm(0);
                                alarmEnabled = FALSE;
                                statistics.n_frames++;

                                int bytes = writeBytesSerialPort(ua_frame_disc,
        ASW_BUF_SIZE);
                                printf("INFO: %d bytes written - UA\n", bytes);
                                printf("INFO: Successfully disconnected\n");
```

```
                                        if (showStatistics)
                                        {
                                                printf("\nSTATISTICS:\n");
                                                printf("Number of frames sent: 
%d\n", statistics.n_frames);
                                                printf("Number of Timeouts: 
%d\n", statistics.n_timeouts);
                                                printf("Number of frames retransmited: 
%d\n", statistics.n_retransmissions);
                                        }
                                        return 1;
                                        }
                                        else
                                        {
                                        state = START_STATE;
                                        // printf("All the way to the start from BCC\n");
                                        }
                                        break;

                                default:
                                        break;
                                }
                        }
                        }
                }
                printf("TIMEOUT: DISC frame not received!\n");
                break;
                }

                return -1;
        }
```