Faculdade de Engenharia
Universidade do Porto

Data Link Protocol

Computer Networks, Project 1

Class 9 Group 1
Bruno Mendes up201906166@edu.fe.up.pt
José Costa up201907216@edu.fe.up.pt

Wednesday 22nd December, 2021

Contents

Chapter 1

Summary

The project consists of the development of a data link protocol to deliver an error free transfer interface to an agnostic file transfer application.

The developed protocol is tested in a real world usage scenario and is found to be at least 50% efficient at default settings.

Chapter 2

Introduction

Computer networks pose an essential role in today's society. It is essential that network interfaces provide the end user with error free and reliable data transmission abstractions, to assure the integrity of the information being interchanged across devices.

The lower layer of a computer network - the physical medium - is prone to environmental interference, causing errors in the transmitted data, ranging from wrong bits to entire bytes lost. The goal of the upper layer - the data link layer - is to regulate data flow, while reducing transmission errors through mechanisms such as check-sums or parity checks. Higher layers may take advantage of the data link layer interface to broadly distribute data in an organized way across more complex systems than single peer-to-peer.

This report describes the implementation of a peer-to-peer communication protocol to sit above the *RS-232* physical protocol, common in serial ports. To demonstrate a real usage case scenario of the protocol, a simple Linux application to transfer a single file was developed. The application collects information that allows for the study of the data link layer performance and errors related to physical interference.

The next chapter describes the architecture of the project, focusing on the features provided by the developed modules. After that follows a more detailed description of the implementation techniques of the two layers. At the end, a study is made on the validity and efficiency of the protocol.

Chapter 3

Developed modules

3.1 Data link

This module includes the data link layer related functions and provides an interface for the application layer.

3.1.1 *frame* sub module

This sub module provides functions to manipulate frames.

Data structures

aux_frame Auxiliary frame buffer, used for storing temporary frames in the stuffing and destuffing functions.

Exported functions

byte_xor This helper function calculates the exclusive-or operation over an array of bytes. It is used for calculating the BCC fields in frames.

assemble_suframe This function assembles a supervision / unnumbered frame from the role and control fields passed as parameters.

stuff_frame Replaces bytes equal to the flag or the escape character inside the frame by the exclusive-or operation between that byte and 0x20 byte preceded by the escape character, ie. 0x7d 0x5e for the flag and 0x7d 0x5d for the escape character.

destuff_frame Replaces the escaped sequences in stuff bytes by their original bytes.

assemble_iframe Assembles an information frame from the role, control word and destuffed data and size.

read_frame Reads bytes from file and discards until a flag is found. After that reads until a new flag is found, returning the complete frame. Times out if no bytes are read.

3.1.2 *data_link* sub module

This sub module provides the interface for the application layer to receive and send data, handling physical errors and returning error if the resolution is not possible.

Protocol settings

Constants that alter the protocol performance and behavior. Should not be changed in production environments. If one desires to change them for testing purposes, should do so via compiler flags or the provided *Makefile* arguments.

BAUD_RATE The maximum connection speed (bits/s). Available options depend on the system architecture. Defaults to B38400.

CONNECTION_TIMEOUT_TS Time interval before a read operation from a file times out if no bytes are read. Defaults to 5 tenths of a second.

CONNECTION_MAX_TRIES The amount of times functions try to repeat operations before failing, ie. read a frame or expect a valid frame from the other side. Defaults to 60 tries.

INDUCED_FER The probability (0-100) of an information frame validation artificially failing at the receiver side, to simulate frame errors and re-transmissions. Defaults to 0

INDUCED_DELAY_US The amount of time the information frame validation artificially hangs, to test the behavior of the protocol in the case of a slow receiver. Defaults to 0 micro-seconds.

Data structures

old_tio Original port settings for restoration after closing the connection.

connection_role The role of the device in the connection. Used for determining the algorithm for closing the connection, and making sure a reader does not attempt to write data or vice-versa.

in_frame / out_frame Auxiliary buffers for holding read/written frames, respectively. Statically allocated with a fixed size (8192 bytes), to avoid the overhead of dynamic memory management, which means that users are limited to a maximum packet size.

if_error_count Information frame validation errors counter on the receiver side.

Internal functions

restore_close_port Set file descriptor state to *old_tio* settings and closes the file.

read_validate_suf Reads a frame and tests if contents are as expected for a supervisioned/unnumbered frame.

read_validate_if Reads a frame and tests if contents are as expected for a information frame (including the byte check count). Hangs or returns with artificial failure if aforementioned protocol settings are tampered with.

Exported functions

llgeterrors The current value of *if_error_count*.

llopen Opens port, modifies port configuration and executes connection start exchange sequence depending on the device rule. See next chapter for more information.

llwrite Assembles an information frame from the received packet, sends it and waits for acknowledgement.

llread Reads an information frame and acknowledges it to the emitter if it is valid.

llclose Executes connection end exchange sequence depending on the device rule, restores original port configuration and closes port. See next chapter for more information.

3.2 Application

The file transfer application.

3.2.1 *utils* sub module

Exported functions

print_transfer_progress_bar Outputs the progress bar indicating the relative progress of the transfer in file bytes. For optimization reasons, does nothing if the percentage to show did not change, despite a raise in the number of bytes transferred.

elapsed_seconds Elapsed seconds between two time frames, with nanosecond precision.

print_bytes Print array bytes, one by one, in hexadecimal. Useful for debugging.

3.2.2 *packet* sub module

Exported functions

assemble_control_packet Assembles a control packet indicating the start or the end of the transmitted file.

assemble_data_packet Assembles an application layer data packet, containing a portion of the file to send.

3.2.3 *sender* sub module

Data structures

packet Auxiliary buffer to hold the packet to send.

Exported functions

assemble_packet_file_name Assemble the file name to send in a control packet, by user decision, or, if missing, from the path of the file.

send_control_packet Assembles a control packet and sends it using the protocol.

send_file_data Read file, with a user defined step, and send it progressively using the protocol. Show progress bar.

3.2.4 *receiver* sub module

Data structures

packet Auxiliary buffer to hold the packet to read.

bytes_per_packet Read bytes per packet from the control packet. Useful for generating statistics on the receiver side.

file_size Read file size from the control packet. Useful for double checking that the read file has the size it was supposed to have.

Exported functions

get_receiver_bytes_per_packet The *bytes_per_packet* value.

read_validate_control_packet Reads a control packet, retrieving its data and checking for errors. If the control is the end packet, it is compared to the start packet.

write_file_from_stream Read data from the protocol, and output progressively to a file. Show progress bar.

3.2.5 *application* sub module

The application acts as a receiver or a sender of a file, using the protocol in an agnostic way.

Available options

Through command line arguments.

port The port to open.

role_path Send file located in path, or receive to file located in path.

name The file name to send.

bytes_per_packet The size of the data portion of each packet to send.

verbose Whether to show more verbose log messages. Use to view statistics on the receiver side, at the end of the transmission.

Function call sequence

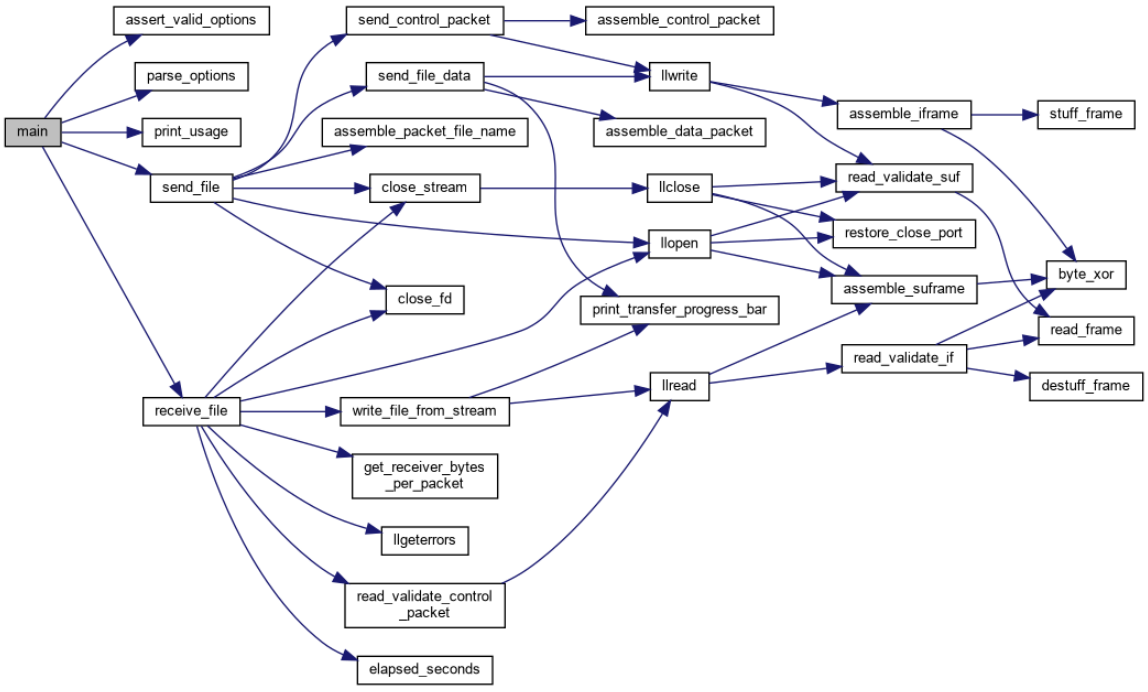

Figure 3.1: Function call sequence graph from *main*. Generated with *Doxygen*.

Chapter 4

Data link protocol

4.1 Framing

At the data link layer, messages are encoded in frames, with a fixed structure.

4.1.1 Supervised/unnumbered frames

Setup procedures and acknowledgements are based on the use of supervised/unnumbered frames:

- **Start Flag** (1 byte): 0x7e; marks the beginning of a frame
- **Address** (1 byte): either transmitter or receiver; resembles the MAC address of more complex protocols
- **Control** (1 byte): the purpose of the message: set connection, disconnect, acknowledge.
- **BBC** (1 byte): Byte Check Count of the first three bytes; ensures header validity
- **End Flag** (1 byte): 0x7e; marks the end of a frame

4.1.2 Information frames

Information flows in information frames, composed of:

- **Start Flag** (1 byte): 0x7e; marks the beginning of a frame
- **Address** (1 byte): either transmitter or receiver; resembles the MAC address of more complex protocols
- **Control** (1 byte): the sequence number of the frame; ensures ordering of the frames
- **BBC1** (1 byte): Byte Check Count of the first three bytes; ensures header validity
- **Data** (several bytes): the actual data to transmit; the packet in upper layers
- **BCC2** (1 byte): Byte Check Count of the data; ensures data validity with a greater degree of confidence the lower the number of data bytes
- **End Flag** (1 byte): 0x7e; marks the end of a frame

Due to the possibility of flag bytes appearing as part of the data or the BCC2, a technique known as *byte stuffing* is applied to the frame before sending it, and its reverse applied after receiving. With the flag being 0x7e and an escape character being 0x7d, substitutions arrive with 0x7d 0x5e for the flag and 0x7d 0x5d for the escape character, effectively enlarging the original data array to at most double its size.

4.2 Connection setup flow

4.2.1 Startup

To startup a connection, a sender must:

- Send SET super-visioned frame
- Wait until receiver sends UA (unnumbered acknowledgement) frame

On the other end, the receiver should logically:

- Wait until a SET frame is received
- Send UA frame

The process is repeated on each end until it is succeeded, or a reasonable timeout is reached.

4.2.2 Disconnect

To end a connection, a sender must:

- Send DISC (disconnect) frame
- Wait until receiver sends DISC frame
- Send UA frame

On the other end, the receiver should logically:

- Wait until a DISC frame is received
- Send DISC frame
- Wait until a UA frame is received

Only senders should initiate a disconnection request. In our implementation, when the sender does not receive acknowledgements from the other side it automatically tries to disconnect, but this was not a mandatory requirement.

4.3 Information transfer flow

In the connection setup, a simple sequence was required, and the whole process could start again if a response is lost or the ordering of the received frames is not as expected. Sending a large number of bytes for the file contents, however, poses a higher challenge, in the way that frames can be lost, considered invalid or arrive at inconstant speeds. The widely proposed solution is the adoption of *Automatic Repeat reQuest (ARQ)* mechanisms, such as:

- **Stop and Wait**: transmit frame and block until positive confirmation is received; timeout and repeat if not received. A sequence number is required to allow the receiver to check if the sent frame is the next to the last it received (and not the same, if positive confirmation was lost and sender sent same frame, in which case the receiver should send a receiver ready for the current frame again and discard read frame).
- **Go Back N (Sliding Window)**: allows transmission of new frames before earlier ones are acknowledged, by the use of an abstracted window that limits the number of packets allowed to be sent without acknowledgement, and advances when they are received.

In this project, we implemented a *Stop and Wait* mechanism with large timeouts, handling connection hazards such as disconnecting the cable. Implementing *Go Back N* would require a more complex flow, aided by two queues and dynamic memory allocation.

Chapter 5

Application protocol

5.1 Packaging

The application is completely agnostic to the data link layer implementation, and assumes data provided by it is error-free. The checks it needs to do are related to the ordering of the frame inside the file to read or to send, and metadata like the file name. At this level, the term *packets* is used to refer to a bunch of data being interchanged.

5.1.1 Control packets

Control packets are sent at the start and the end of the file transfer, and must match:

- **Start/End** (1 byte): signals the beginning or the end of a file transfer
- **Argument type** (1 byte): type of the argument following (file name, size...)
- **Argument size** (1 byte): the size of the argument following in bytes
- **Argument** (size bytes): The argument
- **...**

In our implementation, control packets are used for transmitting the file size, the file name and the bytes sent per packet.

5.1.2 Data packets

Data packets are simply:

- **Control** (1 byte): a control byte (0x1)
- **Sequence number** (1 byte): used for assuring the ordering of the packets
- **Data size** (2 byte, big-endian): the size of the data following in bytes
- **Data** (size bytes): the data

Chapter 6

Protocol efficiency tests

In order to test the performance of the developed protocol and application, several tests were conducted on a real serial port.

To evaluate the performance, the efficiency (S) metric was used. The capacity of the connection is given by $C = 2 * B * log_2(M)$, where B is the sample rate and M is the number of energy levels in the physical medium. In the case of the *RS-232* serial port there are 2 energy levels and $2B$ express the connection baud-rate, therefore $C = Baudrate$. R is the connection speed, given by the division of the total number of transferred bits by the elapsed time in seconds.

The connection efficiency can be calculated using the formula below:

$$S = R/C$$

The tests included varying the port's baud-rate, inducing frame errors on the receiver, manipulating the propagation time and the size of the frames used (via change of the number of bytes in each application packet). With the speeds directly output by the application in verbose mode, the values of S were calculated.

File	Variable	Value	Speed KB/s			Efficiency (S)
pinguim.gif (10968B)	Baud-rate	B38400	3,02	3,02	3,02	0,629
		B19200	1,52	1,52	1,52	0,633
		B57600	4,53	4,53	4,53	0,629
	FER (%)	0	3,02	3,02	3,02	0,629
		5	2,82	2,79	2,80	0,583
		10	2,71	2,60	2,68	0,554
	T_{prop} (us)	0	3,02	3,02	3,02	0,629
		1000	0,12	0,15	0,26	0,037
		500000	0,07	0,05	0,07	0,013
	Bytes per packet	50	2,53	2,53	2,53	0,527
		100	3,02	3,02	3,02	0,629
		200	3,33	3,33	3,33	0,694

Chapter 7

Conclusion

This project aimed to introduce us to the lower level communication mechanisms employed in today's networks, teaching us about how communication flow and error control are handled.

Although simple and designed as an academic example, this protocol makes use of robust methods that ensure the integrity and reliability of the data, such as the Stop and Wait ARQ mechanism, resulting in a protocol efficiency of more than 50% in the majority of cases.

The test application was able to reliably transfer files across devices using an interference prone medium. The file transfer speeds were relatively low to today's standards in modern networks, however expected due to the old nature of the *RS-232* device.

Chapter 8

Addendum

The first version submitted, which we presented to the teacher, did not properly handle repeated sent frames, which caused misbehavior when interference was induced near the serial device, using a metal piece to short circuit communication lines. RR frames sent by the receiver were not received properly by the transmitter, causing it to repeatedly send the same frame, while the receiver was expecting the next one, and responded with REJ to the already received frame.

The code present in the following annex, and also sent as a zip alongside this written report, should behave properly, due to added logic on the *llread* function.

Chapter 9

Annex: source code

9.1 *frame.h*

```
#pragma once

#include <linux/limits.h>
#include <stdbool.h>

/* COMMON TO BOTH FRAMES */
#define F_FLAG 0x7e
#define F_ESCAPE_CHAR 0x7d
#define F_ADDRESS_TRANSMITTER_COMMANDS 0x03
#define F_ADDRESS_RECEIVER_COMMANDS 0x01

/* SUPERVISION/NOT NUMBERED FRAME */
/* F A C BCC1 F */
#define SUF_FRAME_SIZE 5
#define SUF_CONTROL_SET 0x03
#define SUF_CONTROL_DISC 0x0b
#define SUF_CONTROL_UA 0x07
#define SUF_CONTROL_RR(pckt) 0x05 | (pckt << 7)
#define SUF_CONTROL_REJ(pckt) 0x01 | (pckt << 7)

/* INFORMATION FRAME */
/* F A C BCC1 Data BCC2 F */
#define IF_FRAME_SIZE 8192
#define IF_MAX_UNSTUFFED_FRAME_SIZE 4098
#define IF_MAX_DATA_SIZE 4092
#define IF_CONTROL(no_seq) (no_seq << 6)

/**
 * @brief Exclusive or recursively for array of bytes
 *
 * @param data array
 * @param size size of array
 * @return xor
 */
unsigned char byte_xor(unsigned char *data, unsigned size);

/**
 * @brief Stuff flags and escape characters. Frame should be able to hold double
 * the size of the data, in the worst case scenario. Initial and ending flags
 * are not stuffed.
 *
 * @param frame byte array
```

```
 * @param frame_size array size
 * @return stuffed frame size, -1 in case of errors
 */
int stuff_frame(unsigned char *frame, unsigned frame_size);

/**
 * @brief Destuff flags and espace characters from frame.
 *
 * @param frame byte array
 * @param frame_size array size
 * @return unstuffed frame size, -1 in case of errors
 */
int destuff_frame(unsigned char *frame, unsigned frame_size);

/**
 * @brief Build supervisioned/unnumbered frame with given parameters.
 *
 * @param out_frame output buffer
 * @param role transmitter or receiver
 * @param ctr control byte
 */
void assemble_suframe(unsigned char *out_frame, int role, unsigned char ctr);

/**
 * @brief Build information frame with given parameters. Stuffs given frame
 * inside.
 *
 * @param out_frame output buffer
 * @param role receiver or transmitter
 * @param ctr control byte
 * @param unstuffed_data data array
 * @param unstuffed_data_size data array size
 * @return stuffed frame size
 */
int assemble_iframe(unsigned char *out_frame, int role, unsigned char ctr,
                    unsigned char *unstuffed_data,
                    unsigned unstuffed_data_size);

/**
 * @brief Read frame from file descriptor. Discards all bytes until a flag is
 * found. Times out if no bytes read.
 *
 * @param out_frame output buffer
 * @param max_frame_size output buffer size
 * @param fd file to read
 * @return read frame size, -1 if timeout or error
 */
int read_frame(unsigned char *out_frame, unsigned max_frame_size, int fd);
```

9.2 *frame.c*

```
#include "frame.h"
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>
#include <termios.h>
#include <unistd.h>

static char aux_frame[IF_FRAME_SIZE];
```

```
unsigned char byte_xor(unsigned char *data, unsigned size) {
    if (size < 2)
        return 0;
    char mxor = data[0];
    for (int i = 1; i < size; i++) {
        mxor ^= data[i];
    }
    return mxor;
}

int stuff_frame(unsigned char *frame, unsigned frame_size) {
    if (frame_size > IF_MAX_UNSTUFFED_FRAME_SIZE) {
        return -1;
    }

    int new_size = 0;
    for (int i = 0; i < frame_size; i++) {
        bool is_trailing = i == 0 || i == frame_size - 1;
        if (!is_trailing && frame[i] == F_FLAG) {
            aux_frame[new_size++] = F_ESCAPE_CHAR;
            aux_frame[new_size++] = 0x5e;
        } else if (frame[i] == F_ESCAPE_CHAR) {
            aux_frame[new_size++] = F_ESCAPE_CHAR;
            aux_frame[new_size++] = 0x5d;
        } else {
            aux_frame[new_size++] = frame[i];
        }
    }
    memcpy(frame, aux_frame, new_size);
    return new_size;
}

int destuff_frame(unsigned char *frame, unsigned frame_size) {
    int new_size = 0;
    for (int i = 0; i < frame_size; i++) {
        if (frame[i] == F_ESCAPE_CHAR) {
            if (i == frame_size - 1) {
                return -1;
            }
            if (frame[i + 1] == 0x5e) {
                aux_frame[new_size++] = F_FLAG;
                i++;
            } else if (frame[i + 1] == 0x5d) {
                aux_frame[new_size++] = F_ESCAPE_CHAR;
                i++;
            } else {
                return -1;
            }
        } else {
            aux_frame[new_size++] = frame[i];
        }
    }
    memcpy(frame, aux_frame, new_size);
    return new_size;
}

void assemble_suframe(unsigned char *out_frame, int role, unsigned char ctr) {

    out_frame[0] = F_FLAG;
    out_frame[1] = role == 0 ? F_ADDRESS_TRANSMITTER_COMMANDS
                             : F_ADDRESS_RECEIVER_COMMANDS;
    out_frame[2] = ctr;
```

```
    unsigned char fields[] = {out_frame[1], out_frame[2]};
    out_frame[3] = byte_xor(fields, sizeof fields);

    out_frame[4] = F_FLAG;
}

int assemble_iframe(unsigned char *out_frame, int role, unsigned char ctr,
                    unsigned char *unstuffed_data,
                    unsigned unstuffed_data_size) {
    out_frame[0] = F_FLAG;
    out_frame[1] = role == 0 ? F_ADDRESS_TRANSMITTER_COMMANDS
                             : F_ADDRESS_RECEIVER_COMMANDS;
    out_frame[2] = ctr;

    unsigned char fields[] = {out_frame[1], out_frame[2]};
    out_frame[3] = byte_xor(fields, sizeof fields);

    char bcc = byte_xor(unstuffed_data, unstuffed_data_size);

    memcpy(out_frame + 4, unstuffed_data, unstuffed_data_size);

    out_frame[4 + unstuffed_data_size] = bcc;
    out_frame[5 + unstuffed_data_size] = F_FLAG;

    int stuffed_frame_size = stuff_frame(out_frame, 6 + unstuffed_data_size);
    return stuffed_frame_size;
}

int read_frame(unsigned char *out_frame, unsigned max_frame_size, int fd) {
    for (int i = 0; i < max_frame_size;) {
        if (read(fd, out_frame + i, 1) <= 0) {
            break;
        }
        if (i == 0 && out_frame[i] != F_FLAG) {
            continue;
        }
        if (i != 0 && out_frame[i] == F_FLAG) {
            tcflush(fd, TCIFLUSH);
            return i + 1;
        }
        i++;
    }
    tcflush(fd, TCIFLUSH);
    return -1;
}
```

9.3 *data_link.h*

```
#pragma once

#include "frame.h"

/**
 * @brief Roles available for a device
 *
 */
typedef enum device_role { TRANSMITTER, RECEIVER } device_role;

/**
 * @brief Maximum size of an application layer packer
 *
 */
```

```c
#define MAX_PACKET_SIZE 4092

/**
 * @brief Maximum size of the data field of an application layer packet
 *
 */
#define MAX_DATA_PER_PACKET_SIZE 4088

/**
 * @brief Open a data link on /dev/ttyS<port> with given role.
 *
 * @param port of /dev/ttyS<port>
 * @param role one of TRANSMITTER, RECEIVER
 * @return opened file descriptor, -1 in case of error
 */
int llopen(int port, device_role role);

/**
 * @brief Write packet with given length to a previously opened data link.
 *
 * @param fd data link file descriptor
 * @param buffer char character array to transmit
 * @param length buffer length
 * @return number of written chars, -1 in case of error
 */
int llwrite(int fd, unsigned char *buffer, int length);

/**
 * @brief Read packet from a previously opened data link to buffer.
 *
 * @param fd data link file descriptor
 * @param buffer output
 * @return number of read chars, -1 in case of error
 */
int llread(int fd, unsigned char *buffer);

/**
 * @brief Close a previously opened data link.
 *
 * @param fd data link file descriptor
 * @return -1 in case of error, 0 otherwise
 */
int llclose(int fd);

/**
 * @brief Get number of received information frame integrity errors (failed BCC2
 * or invalid headers). Only valid when called from a receiver.
 *
 * @return number of errors
 */
int llgeterrors();
```

9.4 *data_link.c*

```c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <termios.h>
#include <time.h>
#include <unistd.h>
```

```c
#include "data_link.h"
#include "frame.h"

#define NEXT_FRAME_NUMBER(curr) (curr + 1) % 2

/* Protocol settings controllable via compiler flags */
#define DEFAULT_BAUD B38400
#define DEFAULT_TIMEOUT 5
#define DEFAULT_TRIES 60
#define DEFAULT_FER 0
#define DEFAULT_DELAY 0
#ifndef BAUDRATE
#define BAUDRATE DEFAULT_BAUD
#endif
#ifndef CONNECTION_TIMEOUT_TS
#define CONNECTION_TIMEOUT_TS DEFAULT_TIMEOUT
#endif
#ifndef CONNECTION_MAX_TRIES
#define CONNECTION_MAX_TRIES DEFAULT_TRIES
#endif
#ifndef INDUCED_FER
#define INDUCED_FER DEFAULT_FER
#endif
#ifndef INDUCED_DELAY_US
#define INDUCED_DELAY_US DEFAULT_DELAY
#endif

/* Buffers */
static struct termios oldtio;
static device_role connection_role;
static unsigned char in_frame[IF_FRAME_SIZE];
static unsigned char out_frame[IF_FRAME_SIZE];
static unsigned if_error_count = 0;

/**
 * @brief Restore previously changed serial port configuration and close file
 * descriptor.
 *
 * @param fd serial port file descriptor
 */
static void restore_close_port(int fd) {
    if (tcsetattr(fd, TCSADRAIN, &oldtio) == -1) {
        perror("Set termios attributes");
    }
    if (close(fd) == -1) {
        perror("Close stream");
    }
}

/**
 * @brief Blocks until supervisioned/not numbered frame is received and
 * assert content is as expected.
 *
 * @param fd serial port file descriptor
 * @param addr expected address
 * @param cmd expected command
 * @return -1 on read error, -2 on validate error, 0 otherwise
 */
static int read_validate_suf(int fd, unsigned char addr, unsigned char cmd) {
    /* Read frame */
    if (read_frame(in_frame, SUF_FRAME_SIZE, fd) == -1) {
        return -1;
    }
```

```
    /* Validate frame */
    unsigned char expected_suf[SUF_FRAME_SIZE] = {F_FLAG, addr, cmd, addr ^ cmd,
                                                  F_FLAG};
    for (int i = 0; i < SUF_FRAME_SIZE; i++) {
        if (in_frame[i] != expected_suf[i]) {
            return -2;
        }
    }
    return 0;
}

/**
 * @brief Read information frame and assert control bytes are as expected.
 *
 * @param fd serial port file descriptor
 * @param addr expected address
 * @param cmd expected command
 * @param out_data_buffer buffer to write data to (no headers)
 * @return -1 if read error, -2 if header error, -3 if bcc2 error, else read
 * data length
 */
static int read_validate_if(int fd, unsigned char addr, unsigned char cmd,
                            unsigned char *out_data_buffer) {
    /* Read frame */
    int frame_length = -1;
    if ((frame_length = read_frame(in_frame, IF_FRAME_SIZE, fd)) == -1) {
        return -1;
    }

    /* Abort or delay if induced errors are active */
    if (INDUCED_FER > 0) {
        int r = rand() % 100;
        if (r < INDUCED_FER) {
            return -1;
        }
    }
    if (INDUCED_DELAY_US > 0) {
        usleep(INDUCED_DELAY_US);
    }

    /* Validate header */
    unsigned char expected_if_header[4] = {F_FLAG, addr, cmd, addr ^ cmd};
    for (int i = 0; i < 4; i++) {
        if (in_frame[i] != expected_if_header[i]) {
            return -2;
        }
    }

    /* Destuff data and validate BCC2 */
    int unstuffed_frame_length = destuff_frame(in_frame, frame_length);
    if (unstuffed_frame_length == -1) {
        return -1;
    }
    unsigned char bcc2 = in_frame[unstuffed_frame_length - 2];
    if (byte_xor(in_frame + 4, unstuffed_frame_length - 6) != bcc2) {
        return -3;
    }

    /* Copy data to output buffer */
    memcpy(out_data_buffer, in_frame + 4, unstuffed_frame_length - 6);
    return unstuffed_frame_length - 6;
}
```

```
int llgeterrors() {
    return if_error_count;
}

int llopen(int port, device_role role) {
    /* Alert if default protocol settings were changed */
    if (BAUDRATE != DEFAULT_BAUD) {
        printf("Alert: baudrate was changed!\n");
    }
    if (CONNECTION_MAX_TRIES != DEFAULT_TRIES) {
        printf("Alert: max tries were changed to %d!\n", CONNECTION_MAX_TRIES);
    }
    if (CONNECTION_TIMEOUT_TS != DEFAULT_TIMEOUT) {
        printf("Alert: connection timeout was changed to %d ts!\n",
               CONNECTION_TIMEOUT_TS);
    }
    if (INDUCED_FER != DEFAULT_FER) {
        printf("Alert: FER is being induced on the receiver side with %d%% "
               "probability!\n",
               INDUCED_FER);
    }
    if (INDUCED_DELAY_US != DEFAULT_DELAY) {
        printf("Alert: delay is being induced on the receiver side: %d us per "
               "frame process!\n",
               INDUCED_DELAY_US);
    }

    /* Prepare random induced error generation */
    srand(time(NULL));

    /* Assemble device file path */
    char port_path[PATH_MAX];
    int c = snprintf(port_path, PATH_MAX, "/dev/ttyS%d", port);
    if (c < 0 || c > PATH_MAX) {
        return -1;
    }

    /* Open port */
    int fd = open(port_path, O_RDWR | O_NOCTTY);
    if (fd == -1) {
        perror("Open stream");
        return -1;
    }

    /* Save current port configuration for later restoration */
    if (tcgetattr(fd, &oldtio) == -1) {
        perror("Get termios attributes");
        return -1;
    }

    /* Set new port configuration */
    struct termios newtio = oldtio;
    newtio.c_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;
    newtio.c_iflag = IGNPAR;
    newtio.c_oflag = 0;
    newtio.c_lflag = 0;                          /* Non canonical */
    newtio.c_cc[VTIME] = CONNECTION_TIMEOUT_TS; /* Timeout for reads */
    newtio.c_cc[VMIN] = 0; /* Do not block waiting for characters */
    if ((tcflush(fd, TCIOFLUSH) == -1) |
        (tcsetattr(fd, TCSANOW, &newtio) == -1)) {
        perror("Set termios attributes");
        restore_close_port(fd);
```

```
        return -1;
    }

    /* Setup connection */
    connection_role = role;
    for (int i = 0; i < CONNECTION_MAX_TRIES; i++) {
        if (connection_role == TRANSMITTER) {
            assemble_suframe(out_frame, TRANSMITTER, SUF_CONTROL_SET);
            write(fd, out_frame, SUF_FRAME_SIZE);
            if (read_validate_suf(fd, F_ADDRESS_TRANSMITTER_COMMANDS,
                                  SUF_CONTROL_UA) < 0) {
                continue;
            }
            return fd;
        } else {
            if (read_validate_suf(fd, F_ADDRESS_TRANSMITTER_COMMANDS,
                                  SUF_CONTROL_SET) < 0) {
                continue;
            }
            assemble_suframe(out_frame, TRANSMITTER, SUF_CONTROL_UA);
            write(fd, out_frame, SUF_FRAME_SIZE);
            return fd;
        }
    }

    restore_close_port(fd);
    return -1;
}

int llread(int fd, unsigned char *buffer) {
    if (connection_role == TRANSMITTER) {
        return -1;
    }

    static unsigned char curr_frame_number = 0;
    unsigned char next_frame_number = NEXT_FRAME_NUMBER(curr_frame_number);
    for (int tries = 0; tries < CONNECTION_MAX_TRIES; tries++) {
        int c = read_validate_if(fd, F_ADDRESS_TRANSMITTER_COMMANDS,
                                 IF_CONTROL(curr_frame_number), buffer);
        if (c == -1) {
            continue;
        } else if (c == -2) {
            assemble_suframe(out_frame, TRANSMITTER,
                             SUF_CONTROL_RR(curr_frame_number));
            write(fd, out_frame, SUF_FRAME_SIZE);
            continue;
        } else if (c == -3) {
            if_error_count++;
            assemble_suframe(out_frame, TRANSMITTER,
                             SUF_CONTROL_REJ(curr_frame_number));
            write(fd, out_frame, SUF_FRAME_SIZE);
            continue;
        } else {
            assemble_suframe(out_frame, TRANSMITTER,
                             SUF_CONTROL_RR(next_frame_number));
            write(fd, out_frame, SUF_FRAME_SIZE);
            curr_frame_number = next_frame_number;
            return c;
        }
    }
    return -1;
}
```

```
int llwrite(int fd, unsigned char *buffer, int length) {
    if (connection_role == RECEIVER) {
        return -1;
    }
    if (length > IF_MAX_DATA_SIZE) {
        return -1;
    }

    static unsigned char curr_frame_number = 0;
    unsigned char next_frame_number = NEXT_FRAME_NUMBER(curr_frame_number);
    int frame_length = assemble_iframe(
        out_frame, TRANSMITTER, IF_CONTROL(curr_frame_number), buffer, length);

    for (int tries = 0; tries < CONNECTION_MAX_TRIES; tries++) {
        write(fd, out_frame, frame_length);
        if (read_validate_suf(fd, F_ADDRESS_TRANSMITTER_COMMANDS,
                              SUF_CONTROL_RR(next_frame_number)) < 0) {
            continue;
        }
        curr_frame_number = next_frame_number;
        return length;
    }
    return -1;
}

int llclose(int fd) {
    for (int i = 0; i < CONNECTION_MAX_TRIES; i++) {
        if (connection_role == TRANSMITTER) {
            assemble_suframe(out_frame, TRANSMITTER, SUF_CONTROL_DISC);
            write(fd, out_frame, SUF_FRAME_SIZE);
            if (read_validate_suf(fd, F_ADDRESS_TRANSMITTER_COMMANDS,
                                  SUF_CONTROL_DISC) < 0) {
                continue;
            }
            assemble_suframe(out_frame, TRANSMITTER, SUF_CONTROL_UA);
            write(fd, out_frame, SUF_FRAME_SIZE);
        } else {
            if (read_validate_suf(fd, F_ADDRESS_TRANSMITTER_COMMANDS,
                                  SUF_CONTROL_DISC) < 0) {
                continue;
            }
            assemble_suframe(out_frame, TRANSMITTER, SUF_CONTROL_DISC);
            write(fd, out_frame, SUF_FRAME_SIZE);
            if (read_validate_suf(fd, F_ADDRESS_TRANSMITTER_COMMANDS,
                                  SUF_CONTROL_UA) < 0) {
                continue;
            }
        }
        restore_close_port(fd);
        return 0;
    }
    restore_close_port(fd);
    return -1;
}
```

9.5 *packet.h*

```
#pragma once

#define MAX_FILENAME_LENGTH 1024  // Max filename lenght (only filename)
#define MAX_FILE_PATH_LENGTH 3072 // Max file path lenght (without filename)
#define MAX_SEQ_NO 256
```

```c
/* CONTROL PACKET */
#define CP_CONTROL_START 2 // Control packet control field for start packets
#define CP_CONTROL_END 3   // Control packet control field for end packets
#define CP_TYPE_SIZE 0     // Control packet type field for total file size
#define CP_TYPE_FILENAME 1 // Control packet type field for filename
#define CP_TYPE_BYTES_PER_PACKET                                              \
    2 // Control packet type field for data bytes in each packet

/* DATA PACKET */
#define DP_CONTROL 1 // Data packet control field
#define DP_SEQ_NO(x) (x % MAX_SEQ_NO)

/**
 * @brief Assembles an application layer data packet based in the function
 * parameters
 *
 * @param out_packet Pointer to the assembled packet
 * @param seq_no Sequence number of the data in the packet (smaller than
 * MAX_SEQ_NO, use DP_SEQ_NO() to assure this)
 * @param data Pointer to the data to be included in the packet
 * @param data_size Number of data bytes to be included in the packet
 * @return int Size (in bytes) of the assembled data packet
 */
int assemble_data_packet(unsigned char *out_packet, unsigned char seq_no,
                         unsigned char *data, unsigned data_size);

/**
 * @brief Assembles an application layer control packet based in the function
 * parameters
 *
 * @param out_packet Pointer to the assembled packet
 * @param end Boolean flag that indicates if this is the end control packet
 * @param file_size Size of the file to be tranferred
 * @param bytes_per_packet Number of data bytes transported in each packet
 * @param file_name Name of the file to be transferred
 * @return int Size (in bytes) of the assembled control packet
 */
int assemble_control_packet(unsigned char *out_packet, int end,
                            unsigned file_size, unsigned bytes_per_packet,
                            char *file_name);
```

9.6 *packet.c*

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

#include "packet.h"

int assemble_data_packet(unsigned char *out_packet, unsigned char seq_no,
                         unsigned char *data, unsigned data_size) {
    out_packet[0] = DP_CONTROL;
    out_packet[1] = DP_SEQ_NO(seq_no);
    out_packet[2] = data_size / 256;
    out_packet[3] = data_size % 256;
    memcpy((void *)(out_packet + 4), data, data_size);

    return 4 + data_size;
}

int assemble_control_packet(unsigned char *out_packet, int end,
                            unsigned file_size, unsigned bytes_per_packet,
```

```
                           char *file_name) {
    out_packet[0] = end ? CP_CONTROL_END : CP_CONTROL_START;

    unsigned curr_byte = 1;
    /* File size */
    out_packet[curr_byte++] = CP_TYPE_SIZE;
    out_packet[curr_byte++] = sizeof(unsigned);
    memcpy(out_packet + curr_byte, &file_size, sizeof(unsigned));
    curr_byte += sizeof(unsigned);

    /* File name */
    unsigned file_name_size = strlen(file_name) + 1;
    out_packet[curr_byte++] = CP_TYPE_FILENAME;
    memcpy(out_packet + curr_byte, &file_name_size, sizeof(unsigned));
    curr_byte += sizeof(unsigned);
    curr_byte += snprintf((char *)out_packet + curr_byte, file_name_size, "%s",
                          file_name) +
                 1;

    /* Max bytes sent per packet */
    out_packet[curr_byte++] = CP_TYPE_BYTES_PER_PACKET;
    out_packet[curr_byte++] = sizeof(unsigned);
    memcpy(out_packet + curr_byte, &bytes_per_packet, sizeof(unsigned));
    curr_byte += sizeof(unsigned);

    return curr_byte;
}
```

9.7 *utils.h*

```
#pragma once

#include <time.h>

/**
 * @brief Prints the progress bar.
 *
 * @param curr_byte Current byte in the transfer.
 * @param file_size File size (in bytes)
 */
void print_transfer_progress_bar(unsigned curr_byte, unsigned file_size);

/**
 * @brief Elapsed time during the transfer.
 *
 * @param start struct timespec containing info about the tranfer start
 * @param end struct timespec containing info about the tranfer end
 * @return double Elapsed time during the tranfer (in seconds with a decimal
 * place)
 */
double elapsed_seconds(struct timespec *start, struct timespec *end);

/**
 * @brief Prints the content of the bytes in buf.
 *
 * @param buf Pointer to a buffer containing bytes
 * @param size Number of bytes to read
 */
void print_bytes(unsigned char *buf, int size);
```

9.8 *utils.c*

```c
#include <stdio.h>

#include "utils.h"

#define clear_screen() printf("\033[H\033[J")
#define PERCENTAGE_BAR_WIDTH 30

void print_transfer_progress_bar(unsigned curr_byte, unsigned file_size) {
    static int last_percentage = -1;
    float percentage = (float)curr_byte / (float)file_size;
    int percentage_ = (int)(percentage * 100);
    if (last_percentage != percentage_) {
        last_percentage = percentage_;
        printf("\r[");
        int pos = PERCENTAGE_BAR_WIDTH * percentage;
        for (int i = 0; i < PERCENTAGE_BAR_WIDTH; i++) {
            if (i < pos) {
                printf("=");
            } else if (i == pos) {
                printf(">");
            } else {
                printf(" ");
            }
        }
        printf("] %d%% (%u/%uB) ", percentage_, curr_byte, file_size);
    }
}

double elapsed_seconds(struct timespec *start, struct timespec *end) {
    double start_secs_decimal = (double)start->tv_nsec / 1000000000;
    double end_secs_decimal = (double)end->tv_nsec / 1000000000;
    double start_secs = (double)start->tv_sec + start_secs_decimal;
    double end_secs = (double)end->tv_sec + end_secs_decimal;
    return end_secs - start_secs;
}

void print_bytes(unsigned char *buf, int size) {
    printf("size: %d\n", size);
    for (int i = 0; i < size; i++) {
        printf("%x ", buf[i]);
    }
    printf("\n\n");
}
```

9.9 *sender.h*

```c
#pragma once

/**
 * @brief Parses a complete file path to an packet filename field sring. If no
 * filename provided gives the name of the file to send.
 *
 * @param out_packet_file_name Pointer to an empty string buffer
 * @param file_name Name of the file to send
 * @param file_path Path to the file to send
 */
void assemble_packet_file_name(char *out_packet_file_name, char *file_name,
                               char *file_path);

/**
 * @brief Sends a control packet, assembling the control packet and sending it.
 *
```

```
 * @param port_fd Port file descriptor
 * @param file_size Size of the file to be tranferred
 * @param bytes_per_packet Number of data bytes transported in each packet
 * @param packet_file_name Name of the file to be transferred
 * @param is_end Boolean flag that indicates if this is the end control packet
 * @return int -1 in case of error, 0 otherwise
 */
int send_control_packet(int port_fd, unsigned file_size, int bytes_per_packet,
                        char *packet_file_name, int is_end);

/**
 * @brief Sends a file, reading data from file, assembling the data packet and
 * sending it and updating the upload progress bar.
 *
 * @param port_fd Port file descriptor
 * @param fd File to send file descriptor
 * @param bytes_per_packet Number of data bytes transported in each packet
 * @param file_size Size of the file to be tranferred
 * @return int -1 in case of error, 0 otherwise
 */
int send_file_data(int port_fd, int fd, int bytes_per_packet, int file_size);
```

9.10 *sender.c*

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

#include "../data_link/data_link.h"
#include "packet.h"
#include "sender.h"
#include "utils.h"

static unsigned char packet[MAX_PACKET_SIZE];

void assemble_packet_file_name(char *out_packet_file_name, char *file_name,
                               char *file_path) {
    if (strlen(file_name) == 0) {
        char *last_name = strrchr(file_path, '/');
        if (last_name != NULL) {
            strncpy(out_packet_file_name, last_name + 1, PATH_MAX);
        } else {
            strncpy(out_packet_file_name, file_path, PATH_MAX);
        }
    } else {
        strncpy(out_packet_file_name, file_name, PATH_MAX);
    }
}

int send_control_packet(int port_fd, unsigned file_size, int bytes_per_packet,
                        char *packet_file_name, int is_end) {
    int packet_size = assemble_control_packet(
        packet, is_end, file_size, bytes_per_packet, packet_file_name);
    if (llwrite(port_fd, packet, packet_size) < packet_size) {
        fprintf(stderr, "Write control packet failed\n");
        return -1;
    }
    return 0;
}

int send_file_data(int port_fd, int fd, int bytes_per_packet, int file_size) {
```

```
    unsigned char data[MAX_DATA_PER_PACKET_SIZE];
    unsigned char seq_no = 0;
    unsigned curr_byte = 0;
    for (;;) {
        int read_bytes = read(fd, data, bytes_per_packet);
        if (read_bytes < 0) {
            fprintf(stderr, "\nFailed reading file at offset %u\n", curr_byte);
            return -1;
        } else if (read_bytes == 0) {
            break;
        } else {
            int packet_size =
                assemble_data_packet(packet, seq_no, data, read_bytes);
            if (llwrite(port_fd, packet, packet_size) < packet_size) {
                fprintf(stderr, "\nFailed writing packet for file offset %u\n",
                        curr_byte);
                return -1;
            }
            curr_byte += read_bytes;
            seq_no++;
            print_transfer_progress_bar(curr_byte, file_size);
        }
    }
    printf("\n");
    return 0;
}
```

9.11 *receiver.h*

```
#pragma once

/**
 * @brief Reads a control packet, validating it against expected parameters and,
 * in case of the start packet, setting the needed global variables.
 *
 * @param port_fd Port file descriptor
 * @param out_file_name Filename expected
 * @param is_end Boolean flag indicating if this control packet is expected to
 * the the end packet
 * @return int -1 in case of bad incorredct data transmission or unsupported
 * parameters, -2 in case of not being the expected packet (not being the
 * expected (start/end) control packet or being a data packet)
 */
int read_validate_control_packet(int port_fd, char *out_file_name, int is_end);

/**
 * @brief Receives a file, reading data packets, validating its control fields,
 * writing the their data field contents to a file and updating the download
 * progress bar.
 *
 * @param port_fd Port file descriptor
 * @param fd File descriptor
 * @return int -1 in case of error, 0 otherwise
 */
int write_file_from_stream(int port_fd, int fd);

/**
 * @brief Get the receiver bytes per packet object.
 *
 * @return unsigned Number of bytes per packet
 */
unsigned get_receiver_bytes_per_packet();
```

9.12 *receiver.c*

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

#include "../data_link/data_link.h"
#include "packet.h"
#include "receiver.h"
#include "utils.h"

static unsigned char packet[MAX_PACKET_SIZE];
static unsigned bytes_per_packet = -1;
static unsigned file_size = -1;

unsigned get_receiver_bytes_per_packet() {
    return bytes_per_packet;
}

int read_validate_control_packet(int port_fd, char *out_file_name, int is_end) {
    /* Read control packet */
    int packet_length = -1;
    if ((packet_length = llread(port_fd, packet)) == -1) {
        return -1;
    }
    static unsigned char ctrl_packet[MAX_PACKET_SIZE];
    if (!is_end) {
        memcpy(ctrl_packet, packet, packet_length);
    }

    /* Validate arguments */
    unsigned char c = is_end ? CP_CONTROL_END : CP_CONTROL_START;
    if (packet[0] != c) {
        return -2;
    }
    unsigned file_name_size = 0;
    char file_name[PATH_MAX];
    for (int i = 1; i < packet_length;) {
        switch (packet[i]) {
            case CP_TYPE_SIZE:
                memcpy(&file_size, packet + i + 2, packet[i + 1]);
                i += (2 + packet[i + 1]);
                break;
            case CP_TYPE_FILENAME:
                memcpy(&file_name_size, packet + i + 1, sizeof(unsigned));
                i += (1 + sizeof(unsigned));
                i += snprintf(file_name, file_name_size, "%s",
                              (char *)packet + i) +
                     1;
                if (!is_end && out_file_name != NULL) {
                    strncpy(out_file_name, file_name, file_name_size);
                }
                break;
            case CP_TYPE_BYTES_PER_PACKET:
                memcpy(&bytes_per_packet, packet + i + 2, packet[i + 1]);
                i += (2 + packet[i + 1]);
                break;
            default:
                fprintf(stderr, "Unsupported parameter type\n");
                return -1;
        }
    }
```

```
    /* Compare with start packet if end packet */
    if (is_end && memcmp(packet + 1, ctrl_packet + 1, packet_length - 1) != 0) {
        return -1;
    }
    return 0;
}

int write_file_from_stream(int port_fd, int fd) {
    unsigned curr_file_size = 0;
    unsigned char seq_no = 0;
    for (;;) {
        if (llread(port_fd, packet) < 0) {
            fprintf(stderr, "\nFailed reading file at offset %u\n",
                    curr_file_size);
            return -1;
        }
        if (packet[0] != DP_CONTROL) {
            fprintf(stderr, "\nByte control for file at offset %u is wrong\n",
                    curr_file_size);
            return -1;
        }
        if (packet[1] != DP_SEQ_NO(seq_no)) {
            fprintf(stderr,
                    "\nSequence number for file at offset %u is wrong\n",
                    curr_file_size);
            return -1;
        }
        int no_bytes = packet[2] * 256 + packet[3];
        if (write(fd, &packet[4], no_bytes) == -1) {
            perror("\nWrite file");
            return -1;
        }
        curr_file_size += no_bytes;
        print_transfer_progress_bar(curr_file_size, file_size);
        if (curr_file_size >= file_size) {
            break;
        }
        seq_no++;
    }
    printf("\n");
    return 0;
}
```

9.13 *application.c*

```
#include <dirent.h>
#include <errno.h>
#include <fcntl.h>
#include <getopt.h>
#include <limits.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>
#include <sys/stat.h>
#include <unistd.h>

#include "../data_link/data_link.h"
#include "receiver.h"
#include "sender.h"
```

```c
#include "utils.h"

#define PATH_MAX 4096
#define DEFAULT_BYTES_PER_PACKET 100

static struct {
    bool port;
    bool role_path;
    bool name;
    bool bytes_per_packet;
    bool verbose;
} options = {false, false, false, false, false};

static device_role role;
static int port = -1;
static unsigned bytes_per_packet = DEFAULT_BYTES_PER_PACKET;
static char file_name[PATH_MAX / 4];
static char file_path[PATH_MAX / 4];
static int fd = -1;
static int port_fd = -1;

#define verbose_printf                                                  \
    if (options.verbose)                                                \
    printf

#define verbose_fprintf                                                 \
    if (options.verbose)                                                \
    fprintf

/**
 * @brief Closes the file  (Registered with atexit())
 *
 */
static void close_fd() {
    if (close(fd) == -1) {
        perror("Close file");
    }
}

/**
 * @brief Closes the port file (Registered with atexit())
 *
 */
static void close_stream() {
    verbose_printf("Trying to close stream...\n");
    if (llclose(port_fd) == -1) {
        fprintf(stderr, "Failed closing stream\n");
        return;
    }
    verbose_printf("Stream closed\n");
}

/**
 * @brief Sends file, getting info from the file, opening it, opening port,
 * sending the start packet, reading the file and sending it, sending the end
 * control packet and closing the file and port fd's.
 *
 * @param port Port number
 * @return int -1 in case of error, 0 otherwise
 */
int send_file(int port) {
    /* Get file size */
    struct stat st;
```

```
    if (stat(file_path, &st) == -1) {
        perror("Get file info");
        return -1;
    }

    if (st.st_size > (off_t)UINT_MAX) {
        fprintf(stderr, "File size is too large\n");
        return -1;
    }

    /* Open file for reading */
    fd = open(file_path, O_RDONLY);
    if (fd == -1) {
        perror("Open file");
        return -1;
    }
    atexit(close_fd);

    /* Open stream */
    verbose_printf("Trying to open stream on /dev/ttyS%d...\n", port);
    if ((port_fd = llopen(port, TRANSMITTER)) == -1) {
        fprintf(stderr, "Failed opening connection\n");
        return -1;
    }
    verbose_printf("Stream open\n");
    atexit(close_stream);

    /* Send start packet */
    verbose_printf("Sending start packet...\n");
    char packet_file_name[PATH_MAX];
    assemble_packet_file_name(packet_file_name, file_name, file_path);
    if (send_control_packet(port_fd, st.st_size, bytes_per_packet,
                            packet_file_name, 0) == -1) {
        fprintf(stderr, "Failed writing start packet\n");
        return -1;
    }
    verbose_printf("Sent start packet\n");

    /* Send file to stream */
    if (send_file_data(port_fd, fd, bytes_per_packet, st.st_size) == -1) {
        return -1;
    }

    /* Send end packet */
    verbose_printf("Sending end packet...\n");
    if (send_control_packet(port_fd, st.st_size, bytes_per_packet,
                            packet_file_name, 1) == -1) {
        fprintf(stderr, "Failed writing end packet\n");
        return -1;
    }
    verbose_printf("Sent end packet\n");

    return 0;
}

/**
 * @brief Receives a file, opening the port, receiving the start control packet,
 * creating a file with name according to the start packet name field, reading
 * from stream and writing to the file the content of the data field, receiving
 * the end packet and closing the file streams, timing and collecting
 * information (FER, Elapsed time and Average speed) on the file data retrieval
 * process.
 *
```

```
 * @param port Port number
 * @return int -1 in case of error, 0 otherwise
 */
int receive_file(int port) {
    /* Open stream */
    verbose_printf("Trying to open stream on /dev/ttyS%d...\n", port);
    if ((port_fd = llopen(port, RECEIVER)) == -1) {
        fprintf(stderr, "Failed opening connection\n");
        return -1;
    }
    verbose_printf("Stream open\n");
    atexit(close_stream);

    /* Read start packet */
    verbose_printf("Reading start packet...\n");
    if (read_validate_control_packet(port_fd, file_name, 0) != 0) {
        fprintf(stderr, "Start packet is not valid\n");
        return -1;
    }
    verbose_printf("Read start packet\n");

    /* Make new file name if file exists */
    char file_path_[PATH_MAX];
    snprintf(file_path_, PATH_MAX, "%s/%s", file_path, file_name);
    for (int n = 1;; n++) {
        if (access(file_path_, F_OK) == 0) {
            snprintf(file_path_, PATH_MAX, "%s/%d-%s", file_path, n, file_name);
        } else {
            if (n > 1) {
                strncpy(file_name, strrchr(file_path_, '/') + 1, PATH_MAX / 4);
                verbose_printf(
                    "File name already exists, creating new file: %s\n",
                    file_name);
            }
            break;
        }
    }

    /* Open new file with received file name */
    if ((fd = open(file_path_, O_WRONLY | O_CREAT | O_TRUNC,
                   S_IRWXU | S_IRWXG | S_IRWXO)) == -1) {
        perror("Open file for writing");
        return -1;
    }
    atexit(close_fd);

    /* Read from stream and write to file */
    struct timespec start_time, end_time;
    if (clock_gettime(CLOCK_MONOTONIC, &start_time) == -1) {
        perror("Clock get start time");
    }
    if (write_file_from_stream(port_fd, fd) != 0) {
        return -1;
    }
    if (clock_gettime(CLOCK_MONOTONIC, &end_time) == -1) {
        perror("Clock get end time");
    }

    /* Validate end packet */
    verbose_printf("Reading end packet...\n");
    if (read_validate_control_packet(port_fd, NULL, 1) != 0) {
        fprintf(stderr, "End packet is not valid\n");
        return -1;
```

```
    }
    verbose_printf("Read end packet\n");

    /* Print statistics */
    if (options.verbose) {
        struct stat st;
        if (stat(file_path_, &st) == -1) {
            perror("Stat");
        } else {
            int errors = llgeterrors();
            int no_packets = st.st_size / get_receiver_bytes_per_packet();
            float percentage = (float)errors / (float)no_packets;
            int percentage_ = (int)(percentage * 100);
            printf("Error rate: %i%% (%d errors in %d packets)\n", percentage_,
                   errors, no_packets);
            double elapsed_secs = elapsed_seconds(&start_time, &end_time);
            double kbs = ((double)st.st_size / 1000) / elapsed_secs;
            printf("Elapsed time: %.2fs\n", elapsed_secs);
            printf("Average speed: %.2fKB/s\n", kbs);
        }
    }

    return 0;
}

/**
 * @brief Prints application usage help message.
 *
 * @param argv Argv pointer to get the invocation method
 */
static void print_usage(char **argv) {
    printf("Usage: %s [-v] -p <port> -s <filepath> -r <outdirectory> [-n "
           "filename] [-b <dataperpacket>]\n",
           argv[0]);
}

/**
 * @brief Function to parse the command options.
 *
 * @param argc Argument counter
 * @param argv Argument vector
 * @return int -1 in case of error, 0 otherwise
 */
static int parse_options(int argc, char **argv) {
    int opt;
    while ((opt = getopt(argc, argv, ":p:s:r:n:b:v")) != -1) {
        switch (opt) {
            case 'p':
                errno = 0;
                port = atoi(optarg);
                if (errno != 0) {
                    perror("Port must be a valid number\n");
                    return -1;
                }
                options.port = true;
                break;
            case 's':
            case 'r':
                if (options.role_path) {
                    printf("Cannot receive and send at the same time\n");
                    return -1;
                }
                if (strlen(optarg) >= PATH_MAX / 4) {
```

```
                    printf("Path length is too large\n");
                    return -1;
                }
                strncpy(file_path, optarg, PATH_MAX / 4);
                role = opt == 's' ? TRANSMITTER : RECEIVER;
                options.role_path = true;
                break;
            case 'n':
                if (strlen(optarg) >= PATH_MAX / 4) {
                    printf("Name length is too large\n");
                    return -1;
                }
                strncpy(file_name, optarg, PATH_MAX / 4);
                options.name = true;
                break;
            case 'b':
                errno = 0;
                bytes_per_packet = atoi(optarg);
                if (errno != 0) {
                    perror("Bytes per packet must be a valid number\n");
                    return -1;
                }
                options.bytes_per_packet = true;
                break;
            case 'v':
                options.verbose = true;
                break;
            default:
                return -1;
        }
    }
    return 0;
}

/**
 * @brief Validates the options parsed from parse_options()
 *
 * @return int -1 in case of error, 0 otherwise
 */
int assert_valid_options() {
    /* Print immediately */
    setbuf(stdout, NULL);

    /* Validate port */
    if (!options.port) {
        verbose_fprintf(stderr, "Undefined port number\n");
        return -1;
    }

    /* Validate path */
    if (!options.role_path) {
        verbose_fprintf(stderr, "No path provided\n");
        return -1;
    }
    if (strlen(file_path) == 0) {
        verbose_fprintf(stderr, "Empty file path");
        return -1;
    }

    /* Validate bytes per packet */
    if (role == TRANSMITTER) {
        if (options.bytes_per_packet) {
            if (bytes_per_packet <= 0 ||
```

```
                bytes_per_packet > MAX_DATA_PER_PACKET_SIZE) {
                fprintf(stderr, "Invalid bytes per packet\n");
                return -1;
            }
        } else {
            verbose_printf(
                "No bytes per packet value given, using default size: %d "
                "bytes\n",
                DEFAULT_BYTES_PER_PACKET);
        }
    } else {
        if (options.bytes_per_packet) {
            fprintf(
                stderr,
                "Application can't specify bytes per packet when receiving\n");
            return -1;
        }
        if (options.name) {
            fprintf(stderr,
                    "Application can't specify file name when receiving\n");
            return -1;
        }
    }

    return 0;
}

int main(int argc, char **argv) {
    if (parse_options(argc, argv) == -1) {
        print_usage(argv);
        return -1;
    }

    if (assert_valid_options() == -1) {
        print_usage(argv);
        return -1;
    }

    switch (role) {
        case TRANSMITTER:
            if (send_file(port) != 0) {
                return -1;
            }
            printf("Sent file: %s\n", file_path);
            break;
        case RECEIVER:
            if (receive_file(port) != 0) {
                return -1;
            }
            printf("Received file: %s\n", file_name);
            break;
    }
    return 0;
}
```