Redes de Computadores
1st Lab Work

Henrique Sousa
David Magalhães

Sumário

This project has as its goal the implementation of a data link protocol. It also serves the purpose of applying the knowledge acquired during theoretical lessons, mainly about the data link layer and the Stop & Wait Automatic Repeat reQuest.
We learned that the protocol allows for the error-prone serial port cable to be used as a reliable communication mechanism by the upper layers, in this case, the application layer. It is therefore important to have a well implemented protocol that handles stuffing and destuffing, and timeouts, errors or duplicate frames flawlessly.

1. Introduction

The purpose of this project is to write and establish a data link protocol, capable of connecting two different machines through a physical serial port.
The report is divided in several numbered sections following the guide set by the instructors.

2. Design

The designed application has both a receiver and a transmitter mode. Upon invocation, the application (application.c) starts executing the respective protocol, communicating with the link layer (dataProtocol.c, receiver.c, transmitter.c) in order to open, close, send or receive data through the connection.The link layer is in charge of assembling the frames with the right instructions, in order to be transmitted. It handles stuffing of the message and verification of data integrity with protection bytes. It relies on yet another function block which handles the physical communication (physicalProtocol.c). From now on, we will refer to this block as the physical layer. It interprets the bytes received making sure it composed a valid message and informs the link layer according to the information that was received. Additionally, there is a module that allows modification of some program options (options.c) through command line arguments.

3. Code structure

Link layer API:

The following functions are responsible for making the connection between the application layer.
int llopen(int porta, linkType type): Opens a file descriptor for the device /dev/ttySX where X is indicated by porta. The type indicates if it is a transmitter or receiver. Returns the file descriptor.
int llwrite(int fd, u_int8_t *buffer, int length): Writes the bytes in the buffer to the file descriptor. Returns the number of bytes written.

int llread(int fd, u_int8_t *buffer): Reads the bytes to the buffer. Returns the number of bytes read.
int llclose(int fd): Closes the file descriptor and closes the connection with the other end f invoked by the transmitter

Link layer data structures:

linkType: Enumeration that makes the distinction between the transmitter and the receiver at the link layer level.

```
typedef enum{
    RECEIVER,
    TRANSMITTER
} linkType;
```

frame: Keeps the bytes that are to be transmitted or that have been received. Is modified both in the link and physical layers.

```
struct frame{
    u_int8_t frame[MAX_SIZE * 2 + 5 + 2];
    int frameUsedSize;
};
```

linkLayer: Has the information necessary for the link layer to communicate with the application and with the physical layer. Keeps stored the linkType and the frame.

```
struct linkLayer{
    char port[20]; // Device /dev/ttySx
    int fd;
    unsigned int sequenceNumber;   // Frame sequence number: 0, 1
    unsigned int timeout;          // Timeout s
    unsigned int numTransmissions; // Retries on timeout
    struct frame frame;
    linkType type;
};
```

Physical layer API:

The following functions are responsible for making the connection between the link layer and the physical layer.
void writeLinkResponse(struct linkLayer *link): Simply sends the information in the frame within the link through the communication channel
i**nt writeLinkCommand(struct linkLayer *link, u_int8_t A, u_int8_t C):** Sends a command which shall be stored in the frame within the link variable. It expects a given answer and will return 0 after receiving it.
int writeLinkInformation(struct linkLayer *link, u_int8_t A): Send the information in the frame within the link variable. It expects a receiver ready command and will 0 after receiving it.
int readLinkCommand(struct linkLayer *link, u_int8_t A, u_int8_t C): Reads a command to the frame within the link variable. It expects a given command and will return 0 after receiving it.
int readLinkInformation(struct linkLayer *link, u_int8_t A, int *Nr): Reads frames from the communication channel that can be information or command frames. Information frames are stored

in the frame within the link variable and 0 is returned. In case of a command the return value will be different for each.

Global variables

options.c
struct PHYSICAL_OPTIONS OPTIONS: Stores the options the user specified through the command line
dataProtocol.c
struct termios oldtio, newtio: Used to keep and restore the status of the files descriptors used
struct linkLayer link: Stores the information related to the communication channel
physicalProtocol.c
Used to handle the timeout of the other end of the communication
int flag = 1: If 1 means that should write to channel once more
int count = 0: How many times the procedure has timed out
int flagDisc = 0: 1 means that has received a disconnect command and the next timeout should terminate the receiver.

Important functions

application.c

sendFile: Sends the file after reading it from memory
writeFrames: Delegates the writing of all the packets the sending of the file implies
writeFrame: Writes a single packet using the llwrite from link layer
writeInformationFrames: Assembles and writes all the information packets
assembleControlFrame: Assembles the control packet
assembleInformationFrame: Assembles one information packet
readFile: Opens the connection and reads the file packets. Connection will be terminated
saveFile: Saves the file to memory
readFrames: Reads the file packets and keeps the information packets in a buffer to save them into memory. uses the llread from linklayer
readControlFrame: Gets the file size and filename from the control frame
readInformationFrame: Gets the packet data and concatenates it to the buffer
Note: Despite the functions having “frame” in the name they actually refer to the packets. The descriptions are accurate.

transmitter.c

Uses the physical layer API functions to send the information to the receiver
openTransmitter: Opens the connection to the transmitter by sending a set command
writeTransmitter: Assembles a frame with the data that is passed:a protection byte is generated and it stuffs the data with said byte. Writes the frame through the communication channel
closeTransmitter: Closes the communication channel by sending a disconnect command followed by an unnumbered acknowledgement

receiver.c

Uses the physical layer API functions to read the information from the transmitter
openReceiver: Waits for the indication from the transmitter to open the communication channel and acknowledges when it does so
readReceiver: Reads a frame and extracts the packet data by destuffing and verifying the integrity by means of the protection byte. Replies appropriately by asking for the next package or informing the transmitter of corruption in the data. If send the disconnect command handles the closing sequence

4. Main use cases

Receiver mode	**Transmitter mode**
To start the application in receiver mode call: ● ./app.exe <port> [...options] It will call the following functions.	To start the application in transmitter mode call: ● ./app.exe <port> <filename> [...options] It will call the following functions.

```
startReceiverProtocol(){
  readFile(){
    llopen();
    readFrames(){
      Loop{
        llread();}
      }
    }
  saveFile();
}
```

```
startTransmitterProtocol(){
  //reads file from memory
  sendFile(){
    llopen();
    writeFrames(){
      assembleControlFrame();
      writeFrame(){ //start control packet
          llwrite();}
      writeInformationFrames(){
        Loop{
          assembleInformationFrame();
          writeFrame(){ //information packets
            llwrite();}
        }
      }
      writeFrame(){ //end control packet
        llwrite();}
    llclose();}
}
```

5. Data link protocol

To implement the protocol we consider that, when there is data transmission between the receiver and the transmitter it can be a command, a command which expects a given answer, or an information frame, which also expects an answer. The first two just imply assembling the right frame and, in the latter, receiving the right sequence afterwards. For the handling of information frames it is necessary to interpret the command received as it might call for retransmission in case of corruption

or duplication. The main receiver loop will have to, besides information frames, detect the closing of the connection. Therefore, the reading of an information frame also had to be able to interpret commands. Once a byte is read, a state machine is updated in order to know whether or not a frame has been recevied and of what type. Additionally, timeouts are implemented with alarms.

llopen(), openTransmitter(), openReceiver(): According to the link type, the openTransmitter or openReceiver functions will execute. The first will write a command with writeLinkCommand(), whereas the second will receive a command with readLinkCommand(). [Annex 1]

llwrite(), writeTransmitter(): Creates the header for the frame, the protection byte for the data and then stuffs it into the frame itself. Finally sends the information to the receiver by calling writeLinkInformation() and returns the number of bytes written. If there was a timeout it returns -1. [Annex 2]

llread(), readReceiver(): In readReceiver, the program will loop until successfully reading an information frame or a disconnect sequence. First it reads a frame that can either be a command or information frame. If it was a disconnect command, it will send another one with writeLinkResponse and wait for an unnumbered acknowledgment (UA). If it was a UA it will have been successful and return 0, with the data in the buffer argument. If it was a duplicated information frame will ask for the one that is currently being expected by sending a RR command with writeLinkResponse. Otherwise, it will destuff the data, verify its integrity by means of the protection byte. Finally, will ask for the next package by sending a RR command and then return the number of bytes read. [Annex 3]

llclose(), closeTransmitter(): Sends the disconnect message and waits for an identical response by calling writeLinkCommand(); If successful, sends an UA command frame by calling writeLinkResponse, which will be the last one sent. The connection will therefore have been closed. [Annex 4]

writeLinkResponse(): Simply writes the frame passed in the link argument to the file descriptor. [Annex 5]

writeLinkCommand(): First, the function sets an alarm in case the receiver times out and then sends the frame. It then enters a non-blocking reading loop until it receives the expected answer. If it does not receive the right answer it will simply time out. [Annex 6]

writeLinkInformation(): Similar to writeLinkCommand but will interpret answers related information frames. First, the function sets an alarm in case the receiver times out and then sends the frame. It then enters a non-blocking reading loop until it receives a receiver ready (RR) or rejection (REJ) command. In any of these cases the alarm is canceled. It also verifies whether or not the answer is related to the current packet. If it receives a RR asking for the next packet the function will terminate returning the number of bytes written. Otherwise, it will redo the procedure described. [Annex 7]

readLinkCommand(): Reads bytes in a non-blocking reading loop until it receives a sequence which coincides with the arguments given. The address and control fields of the frame shall be equal to the variables A and C, respectively, that are passed as arguments. [Annex 8]

readLinkInformation(): Reads bytes in a non-blocking reading loop until it receives a command or information frame. To obtain an information frame it keeps reading until the second flag byte is read. In that case, it returns 0 if it had the right sequence number and -2 if not. If it receives a disconnect command it returns -1. If it receives a UA it returns -3. After the receiver received disconnect command for the first time, this function will be called with a different A argument. In that case, a timeout will be set. If no UA is received until the timeout expires it will still return as if it had. [Annex 9]

6. Application protocol

For the application protocol, the transmitter and the receiver execute a different logic. The transmitter reads the file from memory and then sends it. To send, it opens the connection and then writes the frames: control packet, then the information packets, then another control packet. It then closes the connection. The writing of the packets is the responsibility of the link layer and it simply implies calling the llwrite() function with the right arguments.
The receiver first opens the connection and then reads packets. The reading is the responsibility of the link layer and it simply implies calling the llread() function. The packet read will be on its arguments. With the packet on the receiver side it is its job to interpret them and copy the data to a main buffer which will hold the file. In the end, it copies the file to memory.
startTransmitterProtocol(): Reads the file from memory into a single buffer. Calls sendFile() which sends the buffer to the receiver. [Annex 10]
sendFile(): Opens the connection with the receiver. Calls writeFrames() which will handle packet logic and send the file. Closes the connection with the receiver. [Annex 11]
writeFrames(): Assembles the control packet and sends it. Calls writeInformationFrames which sends all of the information packets. Writes the control packet again. [Annex 12]
assembleControlFrame(): Allocates the necessary size for the control frame. First copies the information related to the file size by setting the parameter and the field size bytes and then copying the size itself. Then copies the information related to the file name by setting the parameter and the field size bytes and then copying the filename itself. [Annex 13]
writeInformationFrames(): Partitions the file in multiple information frames by calling assembleInformationFrames and keeping track of the current progress through the bufIndex. Writes the assembled frame by calling writeFrame(). [Annex 14]
assembleInformationFrame(): Sets the byte with the packet sequence number, copies the size of the data and then the data to the packet. [Annex 15]
writeFrame(): Writes a single packet by calling llwrite() with the packet being the buf argument and with its size [Annex 16]
startReceiverProtocol(): Reads the file sent by the transmitter by calling readFile(). Then call saveFile() which saves the file to memory. [Annex 17]
readFile(): Opens the connection and then calls readFrames which reads the file from the connection [Annex 18]
readFrames(): Reads multiple packets by calling llread(). First reads the control packet by calling readControlFrame(). Starts reading information packets by calling readInformationFrame() which assembles the file in a single buffer. Verifies that the end control packet corresponds to the first. [Annex 19]
readControlFrame(): Reads the number of bytes for the file size and then copies it to the argument filesize. Reads the number of bytes for the file name, allocates space and then copies it to the allocated buffer. [Annex 20]
readInformationFrame(): Reads and verifies the sequence number of the packet. Reads the number of bytes of the data and copies the amount to the buf argument which will hold the file. [Annex 21]
saveFile(): Opens a file whose name has a predetermined prefix and the name that was passed in the control packet. Writes the file which is stored in the buf variable. [Annex 22]

7. Validation

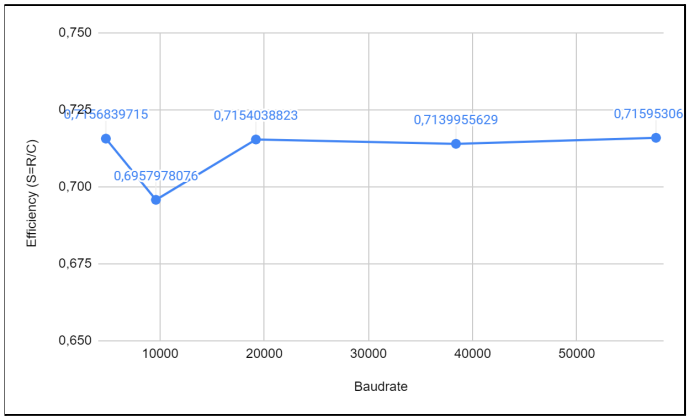
In order to test the validity and efficiency of the program, we ran a series of tests:

- Sending and receiving files of various sizes, most notably a .png file and a .gif file
- Physically interrupting the connection between serial ports and verifying a timeout and a successful transmission afterwards
- Generating errors in the header and data parts of the frame and verifying that a timeout or rejection commands are passed at the link layer level
- Measuring the time it took to transfer a file under different conditions
    - Baudrate Test: Different baudrates
    - FER_HEAD Test: Generating errors in the header of the frame
    - FER_DATA Test: Generating errors in the data section of the frame
    - IFRAME Test: Changing the frame size
    - TPROP Test: Changing the propagation time of the signal.

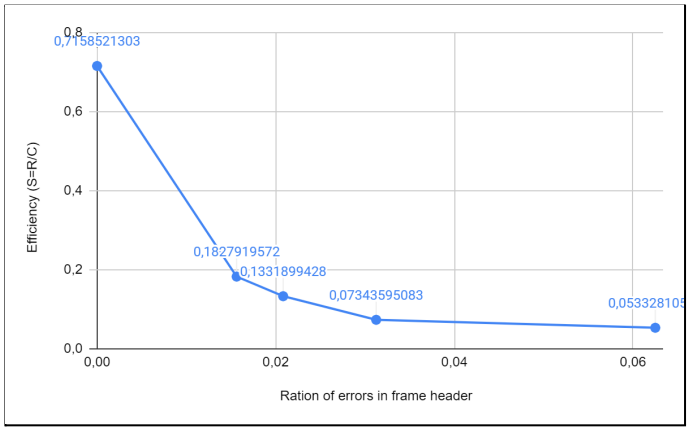
8. Efficiency of data connection protocol

To verify the efficiency of the protocol we ran a series of experiments. We measured the time it took to send between the transmitter and the receiver an image 87744 bits in size. The efficiency will be the ration between the achieved debit (image size dividing by time) and the used baudrate. For the Baudrate Test, the variable in question was the baudrate. Therefore, a constant graph is to be expected. For the other ones, a fixed baudrate of 57600 was used. An increasing or decreasing function are expected.
Theoretically, the expected efficiency would be of 1/(1 + 2*T_PROP), where T_PROP is the time of propagation of the signal. Only the TPROP test varies this parameter being the only one that is expected to follow the formula. Due to the fact that the FER_HEAD, FER_DATA and IFRAME tests focus on measures that affect retransmission, a difference in efficiency will be expected.
The data is available in Annex 23.

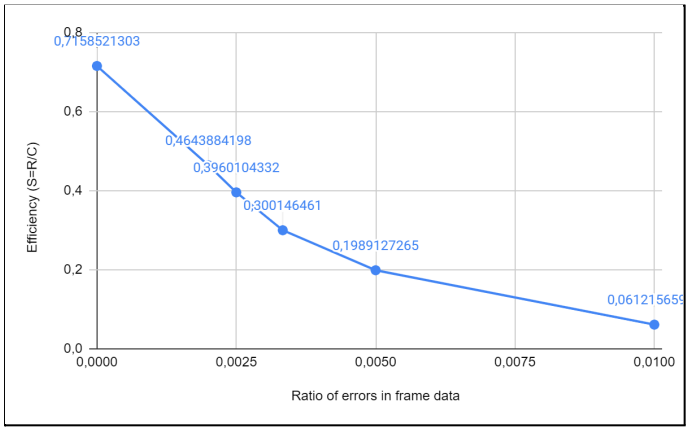
Baudrate Test



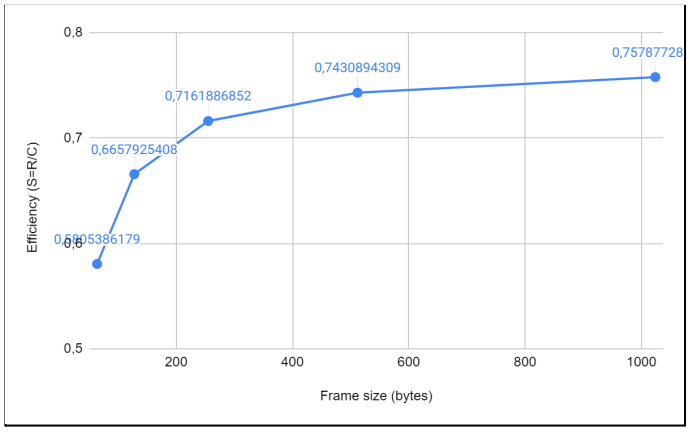
FER_HEAD Test



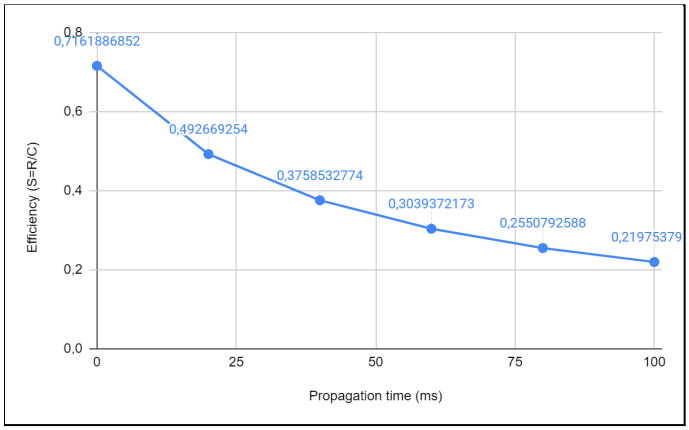
FER_DATA Test



IFRAME Test



TPROP test



9. Conclusion

For the project we were supposed to implement two layers: the application and link layers. We think that this was done successfully and has allowed us to transfer files between two computers as was intended. The design approach worked well and the use of an API to interact between the application and link layer was achieved.

The design of the application layer was functional and simple. There is a sequential logic that is to be followed for the transmitter or receiver. In short, the transmitter reads the file and sends it; and the receiver receives the file and saves it.

The design of the link layer was trickier since accounting for errors in transmission was not trivial. In the end, we were able to handle the rejection of corrupted packages or the request for new ones. The choice of functions at the lowest level and the usage of state machines proved to be a flexible way to manage the flow of the program.

The experiments we performed, as described in sections 8 and 9, allowed us to confirm the proper execution of the program. Additionally, we observed the effects of the variation of different parameters and verified that they agree with the theoretical hypothesis. A change in the baudrate didn't alter the efficiency in any significant way. An increase in the errors or the propagation time decreased the efficiency. An increase in the size of the frame was beneficial to the efficiency. We presume that that is due to the fact that a bigger frame size will result in less frames being transmitted overall and, therefore, less overhead due to the propagation time.

10. Annexes

Annex 1

```
int llopen(int porta, linkType type)
{
    sprintf(link.port, "/dev/ttyS%d", porta);
    setFD(&link, &oldtio, &newtio);
    link.type = type;
    link.sequenceNumber = 0;
    link.timeout = 3;
    link.numTransmissions = 3;
    switch (type)
    {
    case TRANSMITTER:
        if (openTransmitter(&link) == 0)
        {
            return link.fd;
        }
        break;
    case RECEIVER:
        if (openReceiver(&link) == 0)
        {
            return link.fd;
        }
        break;
    }
    return -1;
}

int openTransmitter(struct linkLayer *link)
{

    SETMessage(link->frame.frame);
    link->frame.frameUsedSize = CMDSZ;
    return writeLinkCommand(link, A_EM, UA);
}

int openReceiver(struct linkLayer *link)
{
    int res = readLinkCommand(link, A_EM, SET);
```

```
    if (res == 0)
    {
        UAMessage(link->frame.frame, A_EM);
        link->frame.frameUsedSize = CMDSZ;
        writeLinkResponse(link);
        return 0;
    }
    return -1;
}
```

Annex 2

```
int llwrite(int fd, u_int8_t *buffer, int length)
{
    if (fd != link.fd)
    {
        printf("Wrong fd\n");
    }
    return writeTransmitter(&link, buffer, length);
}

int writeTransmitter(struct linkLayer *link, u_int8_t *buffer, int length)
{
    int res;
    u_int8_t protectionByte;
    IMessage(link->frame.frame, link->sequenceNumber);
    protectionByte = createProtectionByte(buffer,length);
    link->frame.frameUsedSize = stuff(&(link->frame), buffer, length, protectionByte);
    res = writeLinkInformation(link, A_EM);
    if (res == 0)
    {
        link->sequenceNumber = (link->sequenceNumber + 1) % 2; // Changes the sequence
number. N = 2
        return length; // Return number of bytes transmitted
    }
    return -1;
}
```

Annex 3

```
int llread(int fd, u_int8_t *buffer)
{
    if (fd != link.fd)
    {
        printf("Wrong fd\n");
    }
    return readReceiver(&link, buffer);
}

int readReceiver(struct linkLayer *link, u_int8_t *buffer)
{
    int res, Nr, A = A_EM;
    int corrupted = 0, verifyBcc = 0;
    u_int8_t bufferWithBcc[MAX_SIZE + 2];
    int disconnecting = 0;
    while (1)
    {
        corrupted = 0;
        link->frame.frameUsedSize = 0;
        res = readLinkInformation(link, A, &Nr);
        if (res == -1)
        {
            // Received message to disconnect
            DISCMessage(link->frame.frame, A_REC);
            link->frame.frameUsedSize = CMDSZ;
            writeLinkResponse(link);
            A = A_REC;
            disconnecting = 1;
            continue;
        }
        if (res == -2)
        {
            // Received duplicate package. Ask for current package
            RRMessage(link->frame.frame, link->sequenceNumber);
            link->frame.frameUsedSize = CMDSZ;
            writeLinkResponse(link);
            continue;
        }
        if (res == -3)
        {
```

```
            if (disconnecting)
            {
                // Received UA after disconnect
                return 0;
            }
            else
            {
                // Received a random UA. Let's just ignore
                continue;
            }
        }

        // Read successfully the correct package
        // Destuff to a buffer with one extra space for the bcc character
        res = destuff(&(link->frame), bufferWithBcc);
        if (res == -1)
        {
            // Wrong character found after escape character
            corrupted = 1;
        }
        else //Destuffing was ok
        {
            // Verify protection byte
            verifyBcc = verifyProtectionByte(bufferWithBcc, res);
        }

        if (!verifyBcc || corrupted)
        {
            // Was already corrupted or the bcc was wrong
            corrupted = 1;
        }
        else // Protection byte was ok
        {
            // Copy to the buffer return buffer
            res -= 1; // Res included the protection byte. The message is in fact one byte
shorter
            memcpy(buffer, bufferWithBcc, res);
        }
        if (corrupted)
        {
            // There was corruption in the data
            if (Nr != link->sequenceNumber)
```

```
            {
                // Corrupted but it was from a previous package.
                // Ask for current package
                RRMessage(link->frame.frame, link->sequenceNumber);
                link->frame.frameUsedSize = CMDSZ;
                writeLinkResponse(link);
            }
            else
            {
                // The current package was corrupted
                REJMessage(link->frame.frame, link->sequenceNumber);
                link->frame.frameUsedSize = CMDSZ;
                writeLinkResponse(link);
            }

            // Go back to reading
            continue;
        }
        else
        {
            // Ask for next package
            link->sequenceNumber = (link->sequenceNumber + 1) % 2;
            RRMessage(link->frame.frame, link->sequenceNumber);
            link->frame.frameUsedSize = CMDSZ;
            writeLinkResponse(link);
        }
        return res;
    }
}
```

Annex 4

```
int llclose(int fd)
{
    int res;
    switch (link.type)
    {
    case TRANSMITTER:
        res = closeTransmitter(&link);
        break;
    case RECEIVER:
```

```
        break;
    }
    closeFD(fd, oldtio);
    return res;
}
int closeTransmitter(struct linkLayer *link)
{
    int res = 0;
    u_int8_t mes[6];

    DISCMessage(link->frame.frame, A_EM);
    link->frame.frameUsedSize = CMDSZ;

    if (writeLinkCommand(link, A_REC, DISC))
    {
        //Timeout receiving disc
        return -1;
    }

    UAMessage(link->frame.frame, A_REC);
    link->frame.frameUsedSize = CMDSZ;
    writeLinkResponse(link);

    return 0;
}
```

Annex 5

```
void writeLinkResponse(struct linkLayer *link)
{
    int res;
    struct frame frame = link->frame;
    res = FdWrite(link->fd, link->frame.frame, link->frame.frameUsedSize);
    if (res == -1)
    {
        printf("Fd writing error\n");
        exit(1);
    }
}
```

Annex 6

```
int writeLinkCommand(struct linkLayer *link, u_int8_t A, u_int8_t C)
{
    int res;
    struct frame frame = link->frame;
    u_int8_t byte;
    commandState state = START;
    flag = 1;
    count = 0;
    (void)signal(SIGALRM, atende);
    while (count < link->numTransmissions)
    {
        if (flag)
        {
            flag = 0;
            alarm(link->timeout);
            OPTIONS_TPROP();
            res = FdWrite(link->fd, frame.frame, link->frame.frameUsedSize);

            if (res == -1)
            {
                printf("Fd writing error\n");
                exit(1);
            }
        }

        res = read(link->fd, &byte, 1);
        if (res == 0)
        {
            continue;
        }
        if (res == -1)
        {
            printf("Fd reading error \n");
            exit(1);
        }

        state = commandStateMachine(state, A, C, byte);
        if (state == STOP)
        {
            //Received message successful
```

```
            alarm(0);
            return 0;
        }
    }
    return -1; //Couldn't receive an answer Timeout
}
```

Annex 7

```
int writeLinkInformation(struct linkLayer *link, u_int8_t A)
{
    int res, Nr = 0;
    struct frame frame = link->frame;
    u_int8_t byte;
    flag = 1;
    writeInformationState state = WI_START;

    count = 0;
    (void)signal(SIGALRM, atende);
    while (count < link->numTransmissions)
    {
        if (flag)
        {
            flag = 0;
            alarm(link->timeout);
            OPTIONS_TPROP();
            res = FdWrite(link->fd, frame.frame, link->frame.frameUsedSize);

            if (res == -1)
            {
                printf("Fd writing error\n");
                exit(1);
            }
        }

        res = read(link->fd, &byte, 1);
        if (res == 0)
        {
            continue;
        }
        if (res == -1)
```

```
        {
            printf("Fd reading error \n");
            exit(1);
        }

        state = writeInformationStateMachine(state, A, byte, &Nr);
        if (state == WI_STOP_REJ) // Rejection message
        {
            if (Nr == link->sequenceNumber) //Information referring to this frame
            {
                // We will need to retransmit
                alarm(0);         // Cancel scheduled alarm
                count = 0;        // Reset number of attempts because receiver is active
                flag = 1;         // Flag to write again
                state = WI_START; // Set the state to the start
                continue;
            }
            else
            {
                // Probably a delayed answer, but we will retransmit.
                alarm(0);         // Cancel scheduled alarm
                count = 0;        // Reset number of attempts because receiver is active
                flag = 1;         // Flag to write again
                state = WI_START; // Set the state to the start
                continue;
            }
        }
        else if (state == WI_STOP_RR) //Receiver ready message
        {
            if (Nr != link->sequenceNumber) //Is asking for the next frame. All ok
            {
                //Received message successful
                alarm(0);
                return 0;
            }
            else //Duplicate
            {
                // Will retransmit current package
                alarm(0);         // Cancel scheduled alarm
                count = 0;        // Reset number of attempts because receiver is active
                flag = 1;         // Flag to write again
                state = WI_START; // Set the state to the start
```

```
                continue;
            }
        }
    }
    return -1; //Couldn't receive an answer Timeout
}
```

Annex 8

```
int readLinkCommand(struct linkLayer *link, u_int8_t A, u_int8_t C)
{
    int res;
    struct frame frame = link->frame;
    u_int8_t byte;
    commandState state = START;

    while (1)
    {
        res = read(link->fd, &byte, 1);
        if (res == 0)
        {
            continue;
        }
        if (res == -1)
        {
            printf("Fd reading error \n");
            exit(1);
        }
        state = commandStateMachine(state, A, C, byte);
        if (state == STOP)
        {
            //Received command confirmation successfully
            return 0;
        }
    }
}
```

Annex 9

```
int readLinkInformation(struct linkLayer *link, u_int8_t A, int *Nr)
{
```

```
    int res;
    struct frame frame = link->frame;
    u_int8_t byte;
    readInformationState state = RI_START;

    if (A == A_REC)
    {
        // Received message to disconnect. If there is no confirmation will timeout as if
it had
        flagDisc = 0;
        (void)signal(SIGALRM, atendeDisc);
        alarm(link->timeout);
    }
    while (1)
    {
        res = read(link->fd, &byte, 1);

        if (flagDisc)
        {
            flagDisc = 0;
            alarm(0);
            return -3;
        }
        if (res == 0)
        {
            continue;
        }
        if (res == -1)
        {
            printf("Fd reading error \n");
            exit(1);
        }
        if (link->frame.frameUsedSize == MAX_BUFFER_SIZE)
        {
            return -2;
        }
        link->frame.frame[link->frame.frameUsedSize] = byte;
        link->frame.frameUsedSize++;
        state = readInformationStateMachine(state, A, byte, Nr);

        // Generating errors.
        OPTIONS_GENERATE_FER(link, &state, A, Nr);
```

```
        if (state == RI_INFORMATION_STOP)
        {
            if ((*Nr) == link->sequenceNumber)
            {
                // Correct one
                // Received information successfully
                return 0;
            }
            else
            {
                // Duplicate package
                return -2;
            }
        }
        else if (state == RI_READ_STOP)
        {
            // Received a message to disconnect
            return -1;
        }
        else if (state == RI_READ_STOP_UA)
        {
            // Confirm disconenct
            return -3;
        }
        else if (state == RI_RESET)
        {
            link->frame.frameUsedSize = 0; // Reset the size of the buffer
            state = RI_START;
        }
    }
}
```

Annex 10

```
void startTransmitterProtocol(int port, char *filename, int filenameLen)
{
    ssize_t size;
    u_int8_t *buf;
    struct stat st;
    off_t fileSize;
```

```
    int fd = open(filename, O_RDONLY);
    if (fd == -1)
    {
        printf("Error opening file\n");
        exit(1);
    }

    // Get file size and allocate buffer space
    fstat(fd, &st);
    fileSize = st.st_size;
    buf = malloc(fileSize);

    // Read file
    size = read(fd, buf, fileSize);

    sendFile(port, buf, size, filename);
    free(buf);
    return;
}
```

Annex 11

```
void sendFile(int port, u_int8_t *buf, ssize_t size, char *filename)
{
    printf("Starting transmitter protocol\n");
    int res;
    linkType linkType = TRANSMITTER;
    int fd = llopen(port, linkType);
    if (fd == -1)
    {
        printf("Transmitter: Error establishing connection\n");
        exit(1);
    }

    printf("Connection established\n");
    writeFrames(fd, buf, size, filename);
    printf("Data sent. Closing connection\n");
    res = llclose(fd);
    if (res != 0)
    {
        printf("Transmitter: Couldn't inform receiver to close connection. Manual action
may be necessary\n");
```

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

Annex 12

```
void writeFrames(int fd, u_int8_t *buf, ssize_t size, char *filename)
{
    u_int8_t *controlBuf;
    int controlSize;
    assembleControlFrame(&controlBuf, &controlSize, size, filename);

    // Start control frame
    controlBuf[0] = 2; // For start control
    writeFrame(fd, controlBuf, controlSize);

    // Assemble information and send it
    writeInformationFrames(fd, buf, size);

    // End control frame
    controlBuf[0] = 3; // For end control
    writeFrame(fd, controlBuf, controlSize);
    free(controlBuf);
}
```

Annex 13

```
void assembleControlFrame(u_int8_t **buf, int *size, ssize_t fileSize, char *filename)

{
    (*size) = 1 + 2 + sizeof(fileSize) + 2 + strlen(filename) + 1;
    (*buf) = malloc((*size));               //Will be freed in the caller function
    (*buf)[0] = (u_int8_t)0;                // Control field
    (*buf)[1] = (u_int8_t)0;                // Indicating file size
    (*buf)[2] = (u_int8_t)sizeof(fileSize); // Indicating how many bytes for file size
    memcpy((*buf + 3), (u_int8_t *)&fileSize, sizeof(fileSize)); // File size

    (*buf)[3 + sizeof(fileSize)] = (u_int8_t)1;                                    //
Indicating file name
```

```
    (*buf)[3 + sizeof(fileSize) + 1] = (u_int8_t)strlen(filename);                    //
Indicating how many bytes for filename
    snprintf((*buf) + 3 + sizeof(fileSize) + 2, strlen(filename) + 1, "%s", filename); //
Filename
}
```

Annex 14

```
void writeInformationFrames(int fd, u_int8_t *buf, ssize_t size)
{
    // Writes all of the information frames
    u_int8_t frameBuf[MAX_SIZE];
    u_int16_t datasize = MAX_SIZE - INFORMATION_PACKET_HEAD_SIZE;
    u_int16_t packetDataSize;
    u_int8_t packetSeq = 0;
    ssize_t bufIndex = 0;
    int i = 0;
    while (bufIndex < size)
    {
        // Calculates how many bites needs to send
        if (bufIndex + datasize > size) // Doesn't need to use the whole frame to send the
information
        {
            packetDataSize = size - bufIndex;
        }
        else // Will use all the frame available
        {
            packetDataSize = datasize;
        }

        assembleInformationFrame(buf, bufIndex, frameBuf, packetDataSize, packetSeq);
        writeFrame(fd, frameBuf, INFORMATION_PACKET_HEAD_SIZE + packetDataSize);
        packetSeq = (packetSeq + 1) % 256;
        bufIndex += packetDataSize;
        i++;
    }
}
```

Annex 15

```
void assembleInformationFrame(u_int8_t *buf, ssize_t bufIndex, u_int8_t *frameBuf,
u_int16_t datasize, u_int8_t seq)
{
    frameBuf[0] = 1;
    frameBuf[1] = seq;
    memcpy((frameBuf + 2), (u_int8_t *)&datasize, 2);
    memcpy((frameBuf + 4), (buf + bufIndex), datasize);
    // datasize is not the size of the frameBuf but of the data. Assembling is guaranteed
not to exceed framebuf size
}
```

Annex 16

```
void writeFrame(int fd, u_int8_t *buf, int size)
{
    int res = llwrite(fd, buf, size);
    if (res == -1)
    {
        // Time out
        printf("Transmitter: The receiver timed out\n");
        exit(1);
    }
    if (res != size)
    {
        // Different number of bytes was written for some reason
        printf("Transmitter: Couldn't write full packet\n");
        exit(1);
    }
}
```

Annex 17

```
void startReceiverProtocol(int port)
{
    int res = 0;
    u_int8_t *buf;
    char *filename;
    ssize_t size;
    printf("Starting receiver protocol\n");
    res = readFile(port, &buf, &size, &filename);
    if (res != 0)
```

```
    {
        exit(1);
    }

    printf("Saving file to memory\n");
    res = saveFile(filename, buf, size);
    if (res != 0)
    {
        exit(1);
    }
    free(buf);
    free(filename);
    printf("File saved\n");
}
```

Annex 18

```
int readFile(int port, u_int8_t **buf, ssize_t *size, char **filename)
{
    // Don't forget to allocate space for filename
    linkType linkType = RECEIVER;
    printf("Waiting for connection to be established\n");
    int fd = llopen(port, linkType);
    if (fd == -1)
    {
        printf("Receiver: Error establishing connection\n");
        exit(1);
    }
    printf("Connection established\n");
    readFrames(fd, buf, size, filename);
    printf("File received\n");
    return 0;
}
```

Annex 19

```
void readFrames(int fd, u_int8_t **buf, ssize_t *size, char **filename)
{
    int res;
    int start = 0, end = 0, expecting_end = 0;
    u_int8_t control;
    u_int8_t frameBuf[LINK_LAYER_BUFFER_SIZE], frameBufStart[LINK_LAYER_BUFFER_SIZE];
```

```
    int controlPacketSize = 0;
    ssize_t bufIndex = 0;
    u_int8_t seq = 0;

    while (1)
    {
        res = llread(fd, frameBuf);
        if (res == 0)
        {
            if (start && !end)
            {
                // We received a disconnection message without having received the full
file
                printf("Receiver: Disconnection before operation concluded\n");
                exit(1);
            }
            else if (start && end)
            {
                // Disconnect request successfully received
                return;
            }
            else if (!start && !end)
            {
                printf("Receiver: No file received. Exiting\n");
                return;
            }
            else if (!start && end)
            {
                // We received the end packet before the start packet
                printf("Receiver: Unexpected behaviour detected. Exiting\n");
                exit(1);
            }
        }
        control = frameBuf[0];
        switch (control)
        {
        case 1:
            // Information packet
            if (expecting_end)
            {
                printf("Receiver: No more information packets expected\n");
                exit(1);
```

```
            }

            readInformationFrame(*buf, &bufIndex, frameBuf, &seq);
            if (bufIndex == *size)
            {
                expecting_end = 1;
            }
            break;
        case 2:
            // Start packet
            if (expecting_end)
            {
                // Too many safety measures?
                printf("Receiver: No more information packets expected\n");
                exit(1);
            }
            if (start)
            {
                printf("Receiver: Restarting package reception\n");
                end = 0;
                bufIndex = 0;
                expecting_end = 0;
                seq = 0;
            }
            controlPacketSize = readControlFrame(frameBuf, filename, size);
            (*buf) = malloc(*size); // Alloc space for the file in the buffer
            memcpy(frameBufStart, frameBuf, controlPacketSize); // Keep a copy of the start
packet
            start = 1;
            break;
        case 3:
            // End packet
            if (!start)
            {
                // Received end packet without receiving start packet
                printf("Receiver: Unexpected behaviour detected. Exiting\n");
                exit(1);
            }
            if (!expecting_end)
            {
                // Received end packet without reading the indicated number of bytes
                printf("Receiver: Not expecting end packet\n");
```

```
                exit(1);
            }
            // Compare the start and end packets to verify if the information is the same
            frameBuf[0] = 0;
            frameBufStart[0] = 0;
            if (memcmp(frameBuf, frameBufStart, controlPacketSize) != 0)
            {
                printf("Receiver: Start and end control packet are not matching\n");
                exit(1);
            }
            end = 1;
            break;

        default:
            printf("Receiver: Invalid control packet %02x\n", control);
            exit(1);
        }
    }
}
```

Annex 20

```
int readControlFrame(u_int8_t *buf, char **filename, ssize_t *filesize)
{
    // Byte 0 is already verified to be 2 or 3
    int i = 1; //Control frame iterator
    u_int8_t param, bytes;

    for (int j = 0; j < NUM_CONTROL_PARAMS; j++)
    {
        param = buf[i];
        i++;
        bytes = buf[i];
        i++;
        switch (param)
        {
        case 0:
            // fileSize
            memcpy(filesize, (buf + i), bytes);
            break;
        case 1:
            // filename
```

```
            (*filename) = malloc(bytes);
            memcpy(*filename, (buf + i), bytes);
            break;
        default:
            break;
        }
        i += bytes;
    }
    return i;
}
```

Annex 21

```
void readInformationFrame(u_int8_t *buf, ssize_t *bufIndex, u_int8_t *frameBuf, u_int8_t
*seq)
{
    u_int8_t seqNo = frameBuf[1];
    u_int16_t datasize;
    memcpy(&datasize, frameBuf + 2, 2); // Number of data bytes in the packet
    if (seqNo == (*seq))
    {
        // Correct sequence number for the packet
        (*seq) = ((*seq) + 1) % 256;
        memcpy(buf + (*bufIndex), frameBuf + INFORMATION_PACKET_HEAD_SIZE, datasize);
//Copy data
        (*bufIndex) += datasize;
    }
    else
    {
        // Wrong packet sequence number
        printf("Receiver: Incorrect data packet sequence order.\n");
        exit(1);
    }
    return;
}
```

Annex 22

```
int saveFile(char *filename, u_int8_t *buf, ssize_t size)
{
    char *filename2;
```

```
    int res;

    // Write a prefix "DATA" to the filename
    int filename2size = strlen(filename) + 4 + 1;
    filename2 = malloc(filename2size);
    sprintf(filename2, "DATA%s", filename);

    int fd = open(filename2, O_WRONLY | O_TRUNC | O_CREAT);
    if (fd == -1)
    {
        printf("Receiver: Unable to open output file. Make sure file doesn't exist
already\n");
    }
    free(filename2);

    res = write(fd, buf, size);
    if (res == -1)
    {
        printf("Receiver: Error writing to file\n");
        exit(1);
    }
    return 0;
}
```

Annex 23

image.png - 87744 bits
BAUDRATE
4800: 25.542; 25.542; 25.542
9600: 12.792; 12.803; 13.813
19200: 6.388; 6.388; 6.388
38400: 3.201; 3.200; 3.200
57600: 2.127; 2.128; 2.128

IFRAME (57600 baudrate)
64: 2.624; 2.623; 2.624
128: 2.288; 2.288; 2.288
255: 2.128; 2.127; 2.127
512: 2.050; 2.050; 2.050
1024: 2.010; 2.010; 2.010

TPROP(ms) (57600 baudrate)
0: 2.127; 2.128; 2.127

20: 3.092; 3.092; 3.092
40: 4.053; 4.053; 4.052
60: 5.012; 5.012; 5.012
80: 5.972; 5.972; 5.972
100: 6.932; 6.932; 6.932

FER_HEAD (57600 baudrate)
16: 14.539; 36.267; 23.848
32: 26.952; 14.536; 20.743;
48: 14.540; 11.436; 8.336
64: 14.540; 8.333; 2.128
0: 2.128; 2.128; 2.128

FER_DATA (57600 baudrate)
100: 27.330; 26.412; 31.954
200: 7.127; 7.959; 7.889
300: 4.732; 4.922; 5.572
400: 3.669; 3.957; 3.914
500: 3.607; 3.094; 3.140
0: 2.128; 2.128; 2.128

Annex 24 (application.h)

```
#ifndef APPLICATION_H
#define APPLICATION_H

#include <sys/types.h>

#define LINK_LAYER_BUFFER_SIZE 255 //Needs to be the same as the MAX_SIZE in dataProtocol.h
#define INFORMATION_PACKET_HEAD_SIZE 4
#define NUM_CONTROL_PARAMS 2 // Filesize and filename

int main();

void startTransmitterProtocol(int port, char *filename, int filenameLen);

void startReceiverProtocol(int port);

void sendFile(int port, u_int8_t *buf, ssize_t size, char *filename);

void writeFrames(int fd, u_int8_t *buf, ssize_t size, char *filename);

void writeInformationFrames(int fd, u_int8_t *buf, ssize_t size);
```

```
void assembleControlFrame(u_int8_t **buf, int *size, ssize_t fileSize, char *filename);

void assembleInformationFrame(u_int8_t *buf, ssize_t bufIndex, u_int8_t *frameBuf,
u_int16_t datasize, u_int8_t seq);

void writeFrame(int fd, u_int8_t *buf, int size);

int readFile(int port, u_int8_t **buf, ssize_t *size, char **filename);

int saveFile(char *filename, u_int8_t *buf, ssize_t size);

void readFrames(int fd, u_int8_t **buf, ssize_t *size, char **filename);

int readControlFrame(u_int8_t *buf, char **filename, ssize_t *filesize);

void readInformationFrame(u_int8_t *buf, ssize_t *bufIndex, u_int8_t *frameBuf, u_int8_t
*seq);

#endif //APPLICATION_H
```

Annex 25 (application.c)

```
#include "application.h"

#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <time.h>

#include "dataProtocol.h"
#include "commandMessages.h" //printFrame
#include "options.h"

int main(int argc, char **argv)
{
    int res;
```

```
    int port;
    int argNo;
    time_t seed = time(NULL);
    CREATE_OPTIONS(argc, argv, seed);
    if (argc < 2)
    {
        printf("Call with arguments <port> <filename> [...options] \n");
        exit(1);
    }
    errno = 0;
    port = strtol(argv[1], NULL, 10);
    if (errno != 0)
    {
        // The first argument wasn't a number
        printf("Call with arguments <port> <filename> [...options] \n");
        exit(1);
    }

    for (argNo = 2; argNo < argc; argNo++)
    {
        // Starts in 2 because we already got the port. Only need the files or options now
        if (OPTION_IS_FLAG(argv[argNo]))
        {
            break;
        }
    }

    if (argNo == 2)
    {
        // There was only one argument and the rest was flags or reached the end of the
arguments
        startReceiverProtocol(port);
    }
    else if (argNo == 3)
    {
        // There were two arguments and the rest was flags or reached the end of the
arguments
        startTransmitterProtocol(port, argv[2], strlen(argv[2]));
    }
    else
    {
```

```
        // There were more than two arguments before finding a flag or reaching the end of
the arguments
        printf("Call with arguments <port> <filename> [...options] \n");
        exit(1);
    }
}

void startTransmitterProtocol(int port, char *filename, int filenameLen)
{
    ssize_t size;
    u_int8_t *buf;
    struct stat st;
    off_t fileSize;
    int fd = open(filename, O_RDONLY);
    if (fd == -1)
    {
        printf("Error opening file\n");
        exit(1);
    }

    // Get file size and allocate buffer space
    fstat(fd, &st);
    fileSize = st.st_size;
    printf("File size %d", fileSize);
    buf = malloc(fileSize);

    // Read file
    size = read(fd, buf, fileSize);

    sendFile(port, buf, size, filename);
    free(buf);
    return;
}

void startReceiverProtocol(int port)
{
    int res = 0;
    u_int8_t *buf;
    char *filename;
    ssize_t size;
    printf("Starting receiver protocol\n");
    res = readFile(port, &buf, &size, &filename);
```

```
    if (res != 0)
    {
        exit(1);
    }

    printf("Saving file to memory\n");
    res = saveFile(filename, buf, size);
    if (res != 0)
    {
        exit(1);
    }
    free(buf);
    free(filename);
    printf("File saved\n");
}

void sendFile(int port, u_int8_t *buf, ssize_t size, char *filename)
{
    printf("Starting transmitter protocol\n");
    int res;
    linkType linkType = TRANSMITTER;
    int fd = llopen(port, linkType);
    if (fd == -1)
    {
        printf("Transmitter: Error establishing connection\n");
        exit(1);
    }

    printf("Connection established\n");
    writeFrames(fd, buf, size, filename);
    printf("Data sent. Closing connection\n");
    res = llclose(fd);
    if (res != 0)
    {
        printf("Transmitter: Couldn't inform receiver to close connection. Manual action
may be necessary\n");
    }
    printf("Connection closed\n");
}

void writeFrames(int fd, u_int8_t *buf, ssize_t size, char *filename)
{
```

```
    u_int8_t *controlBuf;
    int controlSize;
    assembleControlFrame(&controlBuf, &controlSize, size, filename);

    // Start control frame
    controlBuf[0] = 2; // For start control
    writeFrame(fd, controlBuf, controlSize);

    // Assemble information and send it
    writeInformationFrames(fd, buf, size);

    // End control frame
    controlBuf[0] = 3; // For end control
    writeFrame(fd, controlBuf, controlSize);
    free(controlBuf);
}

void writeInformationFrames(int fd, u_int8_t *buf, ssize_t size)
{
    // Writes all of the information frames
    u_int8_t frameBuf[MAX_SIZE];
    u_int16_t datasize = MAX_SIZE - INFORMATION_PACKET_HEAD_SIZE;
    u_int16_t packetDataSize;
    u_int8_t packetSeq = 0;
    ssize_t bufIndex = 0;
    int i = 0;
    while (bufIndex < size)
    {
        // Calculates how many bites needs to send
        if (bufIndex + datasize > size) // Doesn't need to use the whole frame to send the
information
        {
            packetDataSize = size - bufIndex;
        }
        else // Will use all the frame available
        {
            packetDataSize = datasize;
        }

        assembleInformationFrame(buf, bufIndex, frameBuf, packetDataSize, packetSeq);
        writeFrame(fd, frameBuf, INFORMATION_PACKET_HEAD_SIZE + packetDataSize);
        packetSeq = (packetSeq + 1) % 256;
```

```
        bufIndex += packetDataSize;
        i++;
    }
}

void assembleControlFrame(u_int8_t **buf, int *size, ssize_t fileSize, char *filename)

{
    (*size) = 1 + 2 + sizeof(fileSize) + 2 + strlen(filename) + 1;
    (*buf) = malloc((*size));               //Will be freed in the caller function
    (*buf)[0] = (u_int8_t)0;                // Control field
    (*buf)[1] = (u_int8_t)0;                // Indicating file size
    (*buf)[2] = (u_int8_t)sizeof(fileSize); // Indicating how many bytes for file size
    memcpy((*buf + 3), (u_int8_t *)&fileSize, sizeof(fileSize)); // File size

    (*buf)[3 + sizeof(fileSize)] = (u_int8_t)1;                                    //
Indicating file name
    (*buf)[3 + sizeof(fileSize) + 1] = (u_int8_t)strlen(filename);                 //
Indicating how many bytes for filename
    snprintf((*buf) + 3 + sizeof(fileSize) + 2, strlen(filename) + 1, "%s", filename); //
Filename
}

void assembleInformationFrame(u_int8_t *buf, ssize_t bufIndex, u_int8_t *frameBuf,
u_int16_t datasize, u_int8_t seq)
{
    frameBuf[0] = 1;
    frameBuf[1] = seq;
    memcpy((frameBuf + 2), (u_int8_t *)&datasize, 2);
    memcpy((frameBuf + 4), (buf + bufIndex), datasize);
    // datasize is not the size of the frameBuf but of the data. Assembling is guaranteed
not to exceed framebuf size
}

void writeFrame(int fd, u_int8_t *buf, int size)
{
    int res = llwrite(fd, buf, size);
    if (res == -1)
    {
        // Time out
        printf("Transmitter: The receiver timed out\n");
        exit(1);
```

```
    }
    if (res != size)
    {
        // Different number of bytes was written for some reason
        printf("Transmitter: Couldn't write full packet\n");
        exit(1);
    }
}

int readFile(int port, u_int8_t **buf, ssize_t *size, char **filename)
{
    // Don't forget to allocate space for filename
    linkType linkType = RECEIVER;
    printf("Waiting for connection to be established\n");
    int fd = llopen(port, linkType);
    if (fd == -1)
    {
        printf("Receiver: Error establishing connection\n");
        exit(1);
    }
    printf("Connection established\n");
    readFrames(fd, buf, size, filename);
    printf("File received\n");
    return 0;
}

int saveFile(char *filename, u_int8_t *buf, ssize_t size)
{
    char *filename2;
    int res;

    // Write a prefix "DATA" to the filename
    int filename2size = strlen(filename) + 4 + 1;
    filename2 = malloc(filename2size);
    sprintf(filename2, "DATA%s", filename);

    int fd = open(filename2, O_WRONLY | O_TRUNC | O_CREAT);
    if (fd == -1)
    {
        printf("Receiver: Unable to open output file. Make sure file doesn't exist
already\n");
    }
```

```
    free(filename2);

    res = write(fd, buf, size);
    if (res == -1)
    {
        printf("Receiver: Error writing to file\n");
        exit(1);
    }
    return 0;
}

void readFrames(int fd, u_int8_t **buf, ssize_t *size, char **filename)
{
    int res;
    int start = 0, end = 0, expecting_end = 0;
    u_int8_t control;
    u_int8_t frameBuf[LINK_LAYER_BUFFER_SIZE], frameBufStart[LINK_LAYER_BUFFER_SIZE];
    int controlPacketSize = 0;
    ssize_t bufIndex = 0;
    u_int8_t seq = 0;

    while (1)
    {
        res = llread(fd, frameBuf);
        if (res == 0)
        {
            if (start && !end)
            {
                // We received a disconnection message without having received the full
file
                printf("Receiver: Disconnection before operation concluded\n");
                exit(1);
            }
            else if (start && end)
            {
                // Disconnect request successfully received
                return;
            }
            else if (!start && !end)
            {
                printf("Receiver: No file received. Exiting\n");
                return;
```

```
            }
            else if (!start && end)
            {
                // We received the end packet before the start packet
                printf("Receiver: Unexpected behaviour detected. Exiting\n");
                exit(1);
            }
        }
        control = frameBuf[0];
        switch (control)
        {
        case 1:
            // Information packet
            if (expecting_end)
            {
                printf("Receiver: No more information packets expected\n");
                exit(1);
            }

            readInformationFrame(*buf, &bufIndex, frameBuf, &seq);
            if (bufIndex == *size)
            {
                expecting_end = 1;
            }
            break;
        case 2:
            // Start packet
            if (expecting_end)
            {
                // Too many safety measures?
                printf("Receiver: No more information packets expected\n");
                exit(1);
            }
            if (start)
            {
                printf("Receiver: Restarting package reception\n");
                end = 0;
                bufIndex = 0;
                expecting_end = 0;
                seq = 0;
            }
            controlPacketSize = readControlFrame(frameBuf, filename, size);
```

```
            (*buf) = malloc(*size); // Alloc space for the file in the buffer
            memcpy(frameBufStart, frameBuf, controlPacketSize); // Keep a copy of the start
packet
            start = 1;
            break;
        case 3:
            // End packet
            if (!start)
            {
                // Received end packet without receiving start packet
                printf("Receiver: Unexpected behaviour detected. Exiting\n");
                exit(1);
            }
            if (!expecting_end)
            {
                // Received end packet without reading the indicated number of bytes
                printf("Receiver: Not expecting end packet\n");
                exit(1);
            }
            // Compare the start and end packets to verify if the information is the same
            frameBuf[0] = 0;
            frameBufStart[0] = 0;
            if (memcmp(frameBuf, frameBufStart, controlPacketSize) != 0)
            {
                printf("Receiver: Start and end control packet are not matching\n");
                exit(1);
            }
            end = 1;
            break;

        default:
            printf("Receiver: Invalid control packet %02x\n", control);
            exit(1);
        }
    }
}

int readControlFrame(u_int8_t *buf, char **filename, ssize_t *filesize)
{
    // Byte 0 is already verified to be 2 or 3
    int i = 1; //Control frame iterator
    u_int8_t param, bytes;
```

```
    for (int j = 0; j < NUM_CONTROL_PARAMS; j++)
    {
        param = buf[i];
        i++;
        bytes = buf[i];
        i++;
        switch (param)
        {
        case 0:
            // fileSize
            memcpy(filesize, (buf + i), bytes);
            break;
        case 1:
            // filename
            (*filename) = malloc(bytes);
            memcpy(*filename, (buf + i), bytes);
            break;
        default:
            break;
        }
        i += bytes;
    }
    return i;
}

void readInformationFrame(u_int8_t *buf, ssize_t *bufIndex, u_int8_t *frameBuf, u_int8_t
*seq)
{
    u_int8_t seqNo = frameBuf[1];
    u_int16_t datasize;
    memcpy(&datasize, frameBuf + 2, 2); // Number of data bytes in the packet
    if (seqNo == (*seq))
    {
        // Correct sequence number for the packet
        (*seq) = ((*seq) + 1) % 256;
        memcpy(buf + (*bufIndex), frameBuf + INFORMATION_PACKET_HEAD_SIZE, datasize);
//Copy data
        (*bufIndex) += datasize;
    }
    else
    {
```

```
        // Wrong packet sequence number
        printf("Receiver: Incorrect data packet sequence order.\n");
        exit(1);
    }
    return;
}
```

Annex 26 (byteStuffing.h)

```
#ifndef BYTESTUFFING_H
#define BYTESTUFFING_H
#include "dataProtocol.h"
#include "commandMessages.h" // Instead of #define F 0x7e

#define ESCAPE 0x7d
#define ESCAPE_THE_FLAG 0x5e
#define ESCAPE_THE_ESCAPE 0x5d
#define FIRST_DATA_INDEX 4

#define SPACE_ALLOC_SIZE 20

int stuff(struct frame *frame, u_int8_t tostuff[], int tostuffSize,u_int8_t
protectionByte);

int destuff(struct frame *frame, u_int8_t *buffer);

#endif //BYTESTUFFING_H
```

Annex 27 (byteStuffing.c)

```
#include "byteStuffing.h"

#include <stdlib.h>
#include <string.h>
#include "commandMessages.h"

int stuff(struct frame *frame, u_int8_t tostuff[], int tostuffSize, u_int8_t
protectionByte)
{
    int sfCount = FIRST_DATA_INDEX; //stuffed count
    int nsfCount = 0;               //nonStuffed count
```

```
    int flagged = 0;
    u_int8_t c;

    for (int i = 0; i < tostuffSize; i++)
    {
        c = tostuff[i];
        if (c == ESCAPE)
        {
            frame->frame[sfCount] = ESCAPE;
            frame->frame[sfCount + 1] = ESCAPE_THE_ESCAPE;
            sfCount += 2;
        }
        else if (c == F)
        {
            frame->frame[sfCount] = ESCAPE;
            frame->frame[sfCount + 1] = ESCAPE_THE_FLAG;
            sfCount += 2;
        }
        else
        {
            frame->frame[sfCount] = c;
            sfCount++;
        }
    }

    // Testing the protection byte for a special character
    if (protectionByte == F)
    {

        frame->frame[sfCount] = ESCAPE;
        sfCount++;
        frame->frame[sfCount] = ESCAPE_THE_FLAG;
        sfCount++;
        frame->frame[sfCount] = F;
        sfCount++;
    }
    else if (protectionByte == ESCAPE)
    {
        frame->frame[sfCount] = ESCAPE;
        sfCount++;
        frame->frame[sfCount] = ESCAPE_THE_ESCAPE;
        sfCount++;
```

```
        frame->frame[sfCount] = F;
        sfCount++;
    }
    else
    {
        frame->frame[sfCount] = protectionByte;
        sfCount++;
        frame->frame[sfCount] = F;
        sfCount++;
    }
    return sfCount;
}

int destuff(struct frame *frame, u_int8_t *buffer)
{
    int sfcount = FIRST_DATA_INDEX; //stuffed count
    int nsfCount = 0; //nonStuffed count
    u_int8_t c, protectionByte = 0;

    for (int i = FIRST_DATA_INDEX; frame->frame[i] != F; i++)
    {
        c = frame->frame[i];
        if (c == ESCAPE)
        {
            i++;
            c = frame->frame[i];
            if (c == ESCAPE_THE_FLAG)
            {
                buffer[nsfCount] = F;
            }
            else if (c == ESCAPE_THE_ESCAPE)
            {
                buffer[nsfCount] = ESCAPE;
            }
            else
            { //After an escape character there must always be one of those two
                return -1;
            }
        }
        else
        {
            buffer[nsfCount] = c;
```

```
        }
        nsfCount++;
    }

    return nsfCount;
}
```

Annex 28 (commandMessages.h)

```
#ifndef COMMANDMESSAGES_H
#define COMMANDMESSAGES_H

#include <stdio.h>
#include <sys/types.h>

//COMMANDS
#define A_EM 0x03  //commands by the emissor (or responses by the receiver)
#define A_REC 0x01 //commands by the receiver (or responses by the emissor)
#define F 0x7e
#define SET 0x03
#define DISC 0x0B
#define UA 0x07
#define RR 0x05
#define RR_N1 0x85
#define REJ 0x01
#define REJ_N1 0x81
#define CMDSZ 5 //[FLAG,ADDRESS,CMD,BCC,FLAG]

void printCommand(u_int8_t buf[]);

void printFrame(u_int8_t *buf, int bufLen);

void commandMessage(u_int8_t buf[], u_int8_t A, u_int8_t C);

void SETMessage(u_int8_t buf[]);

void DISCMessage(u_int8_t buf[], u_int8_t A);

void UAMessage(u_int8_t buf[], u_int8_t A);

void RRMessage(u_int8_t buf[], int R);
```

```
void REJMessage(u_int8_t buf[], int R);

void IMessage (u_int8_t buf[], int S);

#endif //COMMANDMESSAGES_H
```

Annex 29 (commandMessages.c)

```
#include "commandMessages.h"

void printCommand(u_int8_t buf[])
{
    printf("Command:");
    for (int i = 0; i < CMDSZ; i++)
    {
        printf("%02x", buf[i]);
    }
    printf("\n");
}

void printFrame(u_int8_t *buf, int bufLen)
{
    printf("Frame: ");
    for (int i = 0; i < bufLen; i++)
    {
        printf("%02x", buf[i]);
    }
    printf("\n");
}

void commandMessage(u_int8_t buf[], u_int8_t A, u_int8_t C)
{
    snprintf(buf, 6, "%c%c%c%c%c", F, A, C, A ^ C, F);
}

void SETMessage(u_int8_t buf[])
{
    commandMessage(buf, A_EM, SET);
}

void DISCMessage(u_int8_t buf[], u_int8_t A)
```

```
{
    commandMessage(buf, A, DISC);
}

void UAMessage(u_int8_t buf[], u_int8_t A)
{
    commandMessage(buf, A, UA);
}

void RRMessage(u_int8_t buf[], int R)
{
    commandMessage(buf, A_EM, RR | (R << 7));
}

void REJMessage(u_int8_t buf[], int R)
{
    commandMessage(buf, A_EM, REJ | (R << 7));
}

void IMessage(u_int8_t buf[], int S){
    u_int8_t C = S << 6;
    snprintf(buf, 5, "%c%c%c%c", F, A_EM, C, A_EM ^ C);
}
```

Annex 30 (dataProtection.h)

```
#ifndef DATA_PROTECTION_H
#define DATA_PROTECTION_H

#include <sys/types.h>

u_int8_t createProtectionByte(u_int8_t *buffer, int length);

int verifyProtectionByte(u_int8_t *buffer, int length);

#endif //DATA_PROTECTION_H
```

Annex 31 (dataProtection.c)

```
#include "dataProtection.h"
```

```
u_int8_t createProtectionByte(u_int8_t *buffer, int length){
    u_int8_t byte = 0;
    for(int i = 0; i < length; i++){
        byte ^= buffer[i];
    }
    return byte;
}

int verifyProtectionByte(u_int8_t *buffer, int length){
    u_int8_t byte = 0;
    for(int i = 0; i< length; i++){
        byte ^= buffer[i];
    }
    return (byte == 0);
}
```

Annex 32 (dataProtocol.h)

```
#ifndef DATAPROTOCOL_H
#define DATAPROTOCOL_H

#include <sys/types.h>

// Change according to frame size. Change application.h define was well
// #define MAX_SIZE 64
// #define MAX_SIZE 128
#define MAX_SIZE 255
// #define MAX_SIZE 512
// #define MAX_SIZE 1024
#define MAX_BUFFER_SIZE MAX_SIZE * 2 + 5 + 2

typedef enum
{
    RECEIVER,
    TRANSMITTER
} linkType;

struct frame
{
    u_int8_t frame[MAX_SIZE * 2 + 5 + 2]; // Frame maximum size
```

```
    // *2 due to the stsuffing. 5 for the commands, 2 for the date proction byte that might
need stuffing as well
    int frameUsedSize;
};

struct linkLayer
{
    char port[20]; // Device /dev/ttySx
    int fd;
    int baudRate;                  // Transmission speed
    unsigned int sequenceNumber;   // Frame sequence number: 0, 1
    unsigned int timeout;          // Timeout s
    unsigned int numTransmissions; // Retries on timeout
    struct frame frame;
    linkType type;
};

int llopen(int porta, linkType type);

int llwrite(int fd, u_int8_t *buffer, int length);

int llread(int fd, u_int8_t *buffer);

int llclose(int fd);

#endif //DATAPROTOCOL_H
```

Annex 33 (dataProtocol.c)

```
#include "dataProtocol.h"

#include <termios.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include "setFD.h"
#include "transmitter.h"
#include "receiver.h"

struct termios oldtio, newtio;
struct linkLayer link;
```

```
int llopen(int porta, linkType type)
{
    sprintf(link.port, "/dev/ttyS%d", porta);
    setFD(&link, &oldtio, &newtio);
    link.type = type;
    link.sequenceNumber = 0;
    link.timeout = 3;
    link.numTransmissions = 3;
    switch (type)
    {
    case TRANSMITTER:
        if (openTransmitter(&link) == 0)
        {
            return link.fd;
        }
        break;
    case RECEIVER:
        if (openReceiver(&link) == 0)
        {
            return link.fd;
        }
        break;
    }
    return -1;
}

int llwrite(int fd, u_int8_t *buffer, int length)
{
    if (fd != link.fd)
    {
        printf("Wrong fd\n");
    }
    return writeTransmitter(&link, buffer, length);
}

int llread(int fd, u_int8_t *buffer)
{
    if (fd != link.fd)
    {
        printf("Wrong fd\n");
    }
    return readReceiver(&link, buffer);
```

```
}

int llclose(int fd)
{
    int res;
    switch (link.type)
    {
    case TRANSMITTER:
        res = closeTransmitter(&link);
        break;
    case RECEIVER:
        break;
    }
    closeFD(fd, oldtio);
    return res;
}
```

Annex 34 (options.h)

```
#ifndef OPTIONS_H
#define OPTIONS_H

#include <time.h>
#include "dataProtocol.h"
#include "physicalProtocol.h"

struct linkLayer;

struct PHYSICAL_OPTIONS
{
    int OPTION_NO_ALARMS;
    int OPTION_FER;
    int OPTION_FER_HEAD;
    int OPTION_FER_DATA;
    int OPTION_TPROP;
    int OPTION_TPROP_MS;
    int OPTIONS_PACKET_LOSS;
    int OPTIONS_PACKET_LOSS_ODD;
};
```

```
void CREATE_OPTIONS(int argc, char **argv, time_t seed);

int OPTION_IS_FLAG(char *arg);

// Error generation is done after receiving a correct information frame
void OPTIONS_GENERATE_FER(struct linkLayer *link, readInformationState *state, u_int8_t A,
int *Nr);

int OPTIONS_ALARM();

void OPTIONS_TPROP();

int OPTIONS_PACKET_LOSS();

#endif //OPTIONS_H
```

Annex 35 (options.c)

```
#include "options.h"

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <time.h> //For FER
#include <errno.h>
#include <unistd.h>
#include "dataProtocol.h"     // for linkLayer
#include "physicalProtocol.h" // for readInformationState

struct PHYSICAL_OPTIONS OPTIONS;

void CREATE_OPTIONS(int argc, char **argv, time_t seed)
{
    errno = 0;
    int flags_started = 0;
    for (int i = 2; i < argc; i++)
    {
        if (!strcmp(argv[i], "-noAlarms"))
        {
            OPTIONS.OPTION_NO_ALARMS = 1;
```

```
            flags_started = 1;
            printf("-noAlarms enabled\n");
        }
        else if (!strcmp(argv[i], "-FER"))
        {
            if (i + 2 >= argc)
            {
                printf("Wrong -FER syntax\n");
                exit(1);
            }
            OPTIONS.OPTION_FER = 1;
            OPTIONS.OPTION_FER_HEAD = (int)strtol(argv[i + 1], NULL, 10);
            OPTIONS.OPTION_FER_DATA = (int)strtol(argv[i + 2], NULL, 10);
            if (errno != 0)
            {
                printf("Passed a wrong value for -FER argument\n");
                exit(1);
            }
            i += 2;
            flags_started = 1;
            printf("-FER enabled with head odds %d and data odds %d\n",
OPTIONS.OPTION_FER_HEAD, OPTIONS.OPTION_FER_DATA);
        }
        else if (!strcmp(argv[i], "-TPROP"))
        {
            if (i + 1 >= argc)
            {
                printf("Wrong -TPROP syntax\n");
                exit(1);
            }
            OPTIONS.OPTION_TPROP = 1;
            OPTIONS.OPTION_TPROP_MS = (int)strtol(argv[i + 1], NULL, 10);
            if (errno != 0)
            {
                printf("Passed a wrong value for -TPROP argument\n");
                exit(1);
            }
            i += 1;
            flags_started = 1;
            printf("-TPROP enabled with %d ms", OPTIONS.OPTION_TPROP_MS);
        }
        else if (!strcmp(argv[i], "-PACKETLOSS"))
```

```
        {
            if (i + 1 >= argc)
            {
                printf("Wrong -PACKETLOSS syntax\n");
                exit(1);
            }
            OPTIONS.OPTIONS_PACKET_LOSS = 1;
            OPTIONS.OPTIONS_PACKET_LOSS_ODD = (int)strtol(argv[i + 1], NULL, 10);
            if (errno != 0)
            {
                printf("Passed a wrong value for -PACKETLOSS argument\n");
                exit(1);
            }
            i += 1;
            flags_started = 1;
            printf("-PACKETLOSS enabled with %d odd", OPTIONS.OPTIONS_PACKET_LOSS_ODD);
        }
        else
        {
            if (flags_started)
            {
                // We have a non flag after a flag
                printf("Call with arguments <port> <filename> [...options]  where <port>
fulfills <condition>\n");
                exit(1);
            }
        }
    }
}

int OPTION_IS_FLAG(char *arg)
{
    if (!strcmp(arg, "-noAlarms"))
    {
        return 1;
    }
    else if (!strcmp(arg, "-FER"))
    {
        return 2;
    }
    else if (!strcmp(arg, "-TPROP"))
    {
```

```
        return 3;
    }
    else if (!strcmp(arg, "-PACKETLOSS"))
    {
        return 4;
    }
    return 0;
}

void OPTIONS_GENERATE_FER(struct linkLayer *link, readInformationState *state, u_int8_t A,
int *Nr)
{
    int r;
    u_int8_t byte;

    if (!OPTIONS.OPTION_FER)
    {
        // Fer is disabled
        return;
    }
    if (*state != RI_INFORMATION_STOP)
    {
        // Fer is only used after receival of a valid information package
        return;
    }

    // Valid information package received. Altering it and returning a new state to
simulate error
    // Errors for head
    if (OPTIONS.OPTION_FER_HEAD != 0)
    {
        for (int i = 1; i <= 3; i++)
        {
            // Head is beteeen indices 1 and 3 inclusively
            r = rand() % OPTIONS.OPTION_FER_HEAD;
            if (!r)
            {
                // 1 in a OPTIONS_FER_DATA chance of being a 0
                // We will totally switch the byte.
                link->frame.frame[i] = link->frame.frame[i] ^ 0xff;
            }
        }
```

```
    }

    // Errors for data
    if (OPTIONS.OPTION_FER_DATA != 0)
    {
        for (int i = 4; i < link->frame.frameUsedSize - 1; i++)
        {
            // Creating errors in the information and bcc2 bytes. Flag is ignored hence -1
            // Keep in mind we are creating errors in the stuffed message. Simulating
errors in transmissions
            r = rand() % OPTIONS.OPTION_FER_DATA;
            if (!r)
            {
                // 1 in a OPTIONS_FER_DATA chance of being a 0
                // We will totally switch the byte.
                link->frame.frame[i] = rand() % 256;
            }
        }
    }

    // Generating new state
    (*state) = RI_START;
    for (int i = 0; i < link->frame.frameUsedSize; i++)
    {
        byte = link->frame.frame[i];
        (*state) = readInformationStateMachine(*state, A, byte, Nr);
        if (*state == RI_INFORMATION_STOP)
        {
            return;
        }
        if (*state == RI_RESET)
        {
            // If reaches a reset means that the errors generated made it an unviable
packet
            // Transmitter will timeout
            return;
        }
    }
}

int OPTIONS_ALARM()
{
```

```
    return (!OPTIONS.OPTION_NO_ALARMS);
}

void OPTIONS_TPROP()
{
    if (OPTIONS.OPTION_TPROP)
    {
        usleep(OPTIONS.OPTION_TPROP_MS * 1000);
    }
}

int OPTIONS_PACKET_LOSS()
{
    if (OPTIONS.OPTIONS_PACKET_LOSS)
    {
        return (rand() % OPTIONS.OPTIONS_PACKET_LOSS_ODD) == 0;
    }
    return 0;
}
```

Annex 36 (physicalProtocol.h)

```
#ifndef PHYSICALPROTOCOL_H
#define PHYSICALPROTOCOL_H

#include "dataProtocol.h"

typedef enum
{
    START,
    FLAG_RCV,
    A_RCV,
    C_RCV,
    BCC_OK,
    STOP
} commandState;

typedef enum
{
    WI_START,
    WI_FLAG_RCV,
```

```
    WI_A_RCV,
    WI_REJ_RCV,
    WI_RR_RCV,
    WI_BCC_REJ_OK,
    WI_BCC_RR_OK,
    WI_STOP_REJ,
    WI_STOP_RR
} writeInformationState;

typedef enum
{
    RI_START,
    RI_FLAG_RCV,
    RI_A_RCV,
    RI_INF,
    RI_DISC,
    RI_UA,
    RI_BCC_DISC_OK,
    RI_BCC_UA_OK,
    RI_INFORMATION_READ,
    RI_INFORMATION_STOP,
    RI_READ_STOP,
    RI_READ_STOP_UA,
    RI_RESET
} readInformationState;

void atende();

int FdWrite(int fd, void *buf, size_t size);

void writeLinkResponse(struct linkLayer *link);

int writeLinkCommand(struct linkLayer *link, u_int8_t A, u_int8_t C);

int writeLinkInformation(struct linkLayer *link, u_int8_t A);

int readLinkCommand(struct linkLayer *link, u_int8_t A, u_int8_t C);

int readLinkInformation(struct linkLayer *link, u_int8_t A, int *Nr);

int commandStateMachine(commandState state, u_int8_t A, u_int8_t C, u_int8_t byte);
```

```
int writeInformationStateMachine(writeInformationState state, u_int8_t A, u_int8_t byte,
int *Nr);

int readInformationStateMachine(readInformationState state, u_int8_t A, u_int8_t byte, int
*Nr);

#endif //PHYSICALPROTOCOL_H
```

Annex 37 (physicalProtocol.c)

```
#include "physicalProtocol.h"

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

#include "commandMessages.h"
#include "options.h"

int flag = 1;
int count = 0;
int flagDisc = 0;

void atende() // atende alarme
{
    if (OPTIONS_ALARM())
    {
        printf("Alarme #%d\n", count);
        flag = 1;
        count++;
    }
}

void atendeDisc()
{
    if (OPTIONS_ALARM())
    {
        flagDisc = 1;
    }
}
```

```
int FdWrite(int fd, void *buf, size_t size)
{
    if (OPTIONS_PACKET_LOSS())
    {
        return size; // Returns count because, as far as the transmitter knows, it was
successful
    }
    return write(fd, buf, size);
}

void writeLinkResponse(struct linkLayer *link)
{
    int res;
    struct frame frame = link->frame;
    res = FdWrite(link->fd, link->frame.frame, link->frame.frameUsedSize);
    if (res == -1)
    {
        printf("Fd writing error\n");
        exit(1);
    }
}

int writeLinkCommand(struct linkLayer *link, u_int8_t A, u_int8_t C)
{
    int res;
    struct frame frame = link->frame;
    u_int8_t byte;
    commandState state = START;
    flag = 1;
    count = 0;
    (void)signal(SIGALRM, atende);
    while (count < link->numTransmissions)
    {
        if (flag)
        {
            flag = 0;
            alarm(link->timeout);
            OPTIONS_TPROP();
            res = FdWrite(link->fd, frame.frame, link->frame.frameUsedSize);

            if (res == -1)
            {
```

```
                printf("Fd writing error\n");
                exit(1);
            }
        }

        res = read(link->fd, &byte, 1);
        if (res == 0)
        {
            continue;
        }
        if (res == -1)
        {
            printf("Fd reading error \n");
            exit(1);
        }

        state = commandStateMachine(state, A, C, byte);
        if (state == STOP)
        {
            //Received message successful
            alarm(0);
            return 0;
        }
    }
    return -1; //Couldn't receive an answer Timeout
}

int writeLinkInformation(struct linkLayer *link, u_int8_t A)
{
    int res, Nr = 0;
    struct frame frame = link->frame;
    u_int8_t byte;
    flag = 1;
    writeInformationState state = WI_START;

    count = 0;
    (void)signal(SIGALRM, atende);
    while (count < link->numTransmissions)
    {
        if (flag)
        {
            flag = 0;
```

```
            alarm(link->timeout);
            OPTIONS_TPROP();
            res = FdWrite(link->fd, frame.frame, link->frame.frameUsedSize);

            if (res == -1)
            {
                printf("Fd writing error\n");
                exit(1);
            }
        }

        res = read(link->fd, &byte, 1);
        if (res == 0)
        {
            continue;
        }
        if (res == -1)
        {
            printf("Fd reading error \n");
            exit(1);
        }

        state = writeInformationStateMachine(state, A, byte, &Nr);
        if (state == WI_STOP_REJ) // Rejection message
        {
            if (Nr == link->sequenceNumber) //Information referring to this frame
            {
                // We will need to retransmit
                alarm(0);         // Cancel scheduled alarm
                count = 0;        // Reset number of attempts because receiver is active
                flag = 1;         // Flag to write again
                state = WI_START; // Set the state to the start
                continue;
            }
            else
            {
                // Probably a delayed answer, but we will retransmit.
                alarm(0);         // Cancel scheduled alarm
                count = 0;        // Reset number of attempts because receiver is active
                flag = 1;         // Flag to write again
                state = WI_START; // Set the state to the start
                continue;
```

```
            }
        }
        else if (state == WI_STOP_RR) //Receiver ready message
        {
            if (Nr != link->sequenceNumber) //Is asking for the next frame. All ok
            {
                //Received message successful
                alarm(0);
                return 0;
            }
            else //Duplicate
            {
                // Will retransmit current package
                alarm(0);         // Cancel scheduled alarm
                count = 0;        // Reset number of attempts because receiver is active
                flag = 1;         // Flag to write again
                state = WI_START; // Set the state to the start
                continue;
            }
        }
    }
    return -1; //Couldn't receive an answer Timeout
}

int readLinkCommand(struct linkLayer *link, u_int8_t A, u_int8_t C)
{
    int res;
    struct frame frame = link->frame;
    u_int8_t byte;
    commandState state = START;

    while (1)
    {
        res = read(link->fd, &byte, 1);
        if (res == 0)
        {
            continue;
        }
        if (res == -1)
        {
            printf("Fd reading error \n");
            exit(1);
```

```
        }
        state = commandStateMachine(state, A, C, byte);
        if (state == STOP)
        {
            //Received command confirmation successfully
            return 0;
        }
    }
}

int readLinkInformation(struct linkLayer *link, u_int8_t A, int *Nr)
{
    int res;
    struct frame frame = link->frame;
    u_int8_t byte;
    readInformationState state = RI_START;

    if (A == A_REC)
    {
        // Received message to disconnect. If there is no confirmation will timeout as if
it had
        flagDisc = 0;
        (void)signal(SIGALRM, atendeDisc);
        alarm(link->timeout);
    }
    while (1)
    {
        res = read(link->fd, &byte, 1);

        if (flagDisc)
        {
            flagDisc = 0;
            alarm(0);
            return -3;
        }
        if (res == 0)
        {
            continue;
        }
        if (res == -1)
        {
            printf("Fd reading error \n");
```

```
            exit(1);
        }
        if (link->frame.frameUsedSize == MAX_BUFFER_SIZE)
        {
            return -2;
        }
        link->frame.frame[link->frame.frameUsedSize] = byte;
        link->frame.frameUsedSize++;
        state = readInformationStateMachine(state, A, byte, Nr);

        // Generating errors.
        OPTIONS_GENERATE_FER(link, &state, A, Nr);

        if (state == RI_INFORMATION_STOP)
        {
            if ((*Nr) == link->sequenceNumber)
            {
                // Correct one
                // Received information successfully
                return 0;
            }
            else
            {
                // Duplicate package
                return -2;
            }
        }
        else if (state == RI_READ_STOP)
        {
            // Received a message to disconnect
            return -1;
        }
        else if (state == RI_READ_STOP_UA)
        {
            // Confirm disconenct
            return -3;
        }
        else if (state == RI_RESET)
        {
            link->frame.frameUsedSize = 0; // Reset the size of the buffer
            state = RI_START;
        }
```

```
    }
}

int commandStateMachine(commandState state, u_int8_t A, u_int8_t C, u_int8_t byte)
{
    static u_int8_t protectionByte = 0;
    switch (state)
    {
    case START:
        if (byte == F)
        {
            protectionByte = 0;
            return FLAG_RCV;
        }
        return START;
    case FLAG_RCV:
        protectionByte ^= byte;
        if (byte == A)
        {
            return A_RCV;
        }
        if (byte == F)
        {
            protectionByte = 0;
            return FLAG_RCV;
        }
        return START;
    case A_RCV:
        protectionByte ^= C;
        if (byte == C)
        {
            return C_RCV;
        }
        if (byte == F)
        {
            protectionByte = 0;
            return FLAG_RCV;
        }
        return START;
    case C_RCV:
        protectionByte ^= byte;
        if (protectionByte == 0)
```

```
        {
            return BCC_OK;
        }
        if (byte == F)
        {
            protectionByte = 0;
            return FLAG_RCV;
        }
        return START;
    case BCC_OK:
        if (byte == F)
        {
            protectionByte = 0;
            return STOP;
        }
        return START;
    }
}

int writeInformationStateMachine(writeInformationState state, u_int8_t A, u_int8_t byte,
int *Nr)
{
    static u_int8_t protectionByte = 0;
    switch (state)
    {
    case WI_START:
        if (byte == F)
        {
            protectionByte = 0;
            return WI_FLAG_RCV;
        }
        return WI_START;
    case WI_FLAG_RCV:
        protectionByte ^= byte;
        if (byte == A)
        {
            return WI_A_RCV;
        }
        if (byte == F)
        {
            protectionByte = 0;
            return WI_FLAG_RCV;
```

```
        }
        return WI_START;
    case WI_A_RCV:
        (*Nr) = byte >> 7;

        protectionByte ^= byte;
        if (byte == REJ || byte == REJ_N1)
        {
            return WI_REJ_RCV;
        }
        else if (byte == RR || byte == RR_N1)
        {
            return WI_RR_RCV;
        }
        if (byte == F)
        {
            protectionByte = 0;
            return WI_FLAG_RCV;
        }
        return WI_START;
    case WI_REJ_RCV:
        protectionByte ^= byte;
        if (protectionByte == 0)
        {
            return WI_BCC_REJ_OK;
        }
        if (byte == F)
        {
            protectionByte = 0;
            return WI_FLAG_RCV;
        }
        return WI_START;
    case WI_RR_RCV:
        protectionByte ^= byte;
        if (protectionByte == 0)
        {
            return WI_BCC_RR_OK;
        }
        if (byte == F)
        {
            protectionByte = 0;
            return WI_FLAG_RCV;
```

```
        }
        return WI_START;
    case WI_BCC_REJ_OK:
        if (byte == F)
        {
            return WI_STOP_REJ;
        }
        return WI_START;
    case WI_BCC_RR_OK:
        if (byte == F)
        {
            return WI_STOP_RR;
        }
        return WI_START;
    }
}

int readInformationStateMachine(readInformationState state, u_int8_t A, u_int8_t byte, int
*Nr)
{
    static u_int8_t protectionByte = 0;

    switch (state)
    {
    case RI_START:
        if (byte == F)
        {
            protectionByte = 0;
            return RI_FLAG_RCV;
        }
        return RI_START;
    case RI_FLAG_RCV:
        if (byte == A_REC || byte == A_EM)
        {
            protectionByte ^= byte;
            return RI_A_RCV;
        }
        if (byte == F)
        {
            protectionByte = 0;
            return RI_FLAG_RCV;
        }
```

```
        return RI_RESET;
    case RI_A_RCV:
        (*Nr) = byte >> 6;
        protectionByte ^= byte;
        if (byte == 0 || byte == (1 << 6))
        {
            return RI_INF;
        }
        if (byte == DISC)
        {
            return RI_DISC;
        }
        if (byte == UA)
        {
            return RI_UA;
        }
        if (byte == F)
        {
            protectionByte = 0;
            return RI_FLAG_RCV;
        }
        return RI_RESET;
    case RI_INF:
        protectionByte ^= byte;
        if (protectionByte == 0)
        {
            return RI_INFORMATION_READ;
        }
        if (byte == F)
        {
            protectionByte = 0;
            return RI_FLAG_RCV;
        }
        return RI_RESET;
    case RI_DISC:
        protectionByte ^= byte;
        if (protectionByte == 0)
        {
            return RI_BCC_DISC_OK;
        }
        if (byte == F)
        {
```

```
            protectionByte = 0;
            return RI_FLAG_RCV;
        }
        return RI_RESET;
    case RI_UA:
        protectionByte ^= byte;
        if (protectionByte == 0)
        {
            return RI_BCC_UA_OK;
        }
        if (byte == F)
        {
            protectionByte = 0;
            return RI_FLAG_RCV;
        }
        return RI_RESET;
    case RI_BCC_DISC_OK:
        if (byte == F)
        {
            return RI_READ_STOP;
        }
        return RI_RESET;
    case RI_BCC_UA_OK:
        if (byte == F)
        {
            return RI_READ_STOP_UA;
        }
        return RI_RESET;
    case RI_INFORMATION_READ:
        if (byte == F)
        {
            return RI_INFORMATION_STOP;
        }
        return RI_INFORMATION_READ;

    default:
        return RI_START;
    }
}
```

Annex 38 (receiver.h)

```
#ifndef RECEIVER_H
#define RECEIVER_H

#include "dataProtocol.h"

int openReceiver(struct linkLayer *link);

int readReceiver(struct linkLayer *link, u_int8_t *buffer);

#endif //RECEIVER_H
```

Annex 39 (receiver.c)

```
#include "receiver.h"

#include <string.h>
#include "commandMessages.h"
#include "physicalProtocol.h"
#include "byteStuffing.h"
#include "dataProtection.h"

int openReceiver(struct linkLayer *link)
{
    int res = readLinkCommand(link, A_EM, SET);
    if (res == 0)
    {
        UAMessage(link->frame.frame, A_EM);
        link->frame.frameUsedSize = CMDSZ;
        writeLinkResponse(link);
        return 0;
    }
    return -1;
}

int readReceiver(struct linkLayer *link, u_int8_t *buffer)
{
    int res, Nr, A = A_EM;
    int corrupted = 0, verifyBcc = 0;
```

```
u_int8_t bufferWithBcc[MAX_SIZE + 2];
int disconnecting = 0;
while (1)
{
    corrupted = 0;
    link->frame.frameUsedSize = 0;
    res = readLinkInformation(link, A, &Nr);
    if (res == -1)
    {
        // Received message to disconnect
        DISCMessage(link->frame.frame, A_REC);
        link->frame.frameUsedSize = CMDSZ;
        writeLinkResponse(link);
        A = A_REC;
        disconnecting = 1;
        continue;
    }
    if (res == -2)
    {
        // Received duplicate package. Ask for current package
        RRMessage(link->frame.frame, link->sequenceNumber);
        link->frame.frameUsedSize = CMDSZ;
        writeLinkResponse(link);
        continue;
    }
    if (res == -3)
    {
        if (disconnecting)
        {
            // Received UA after disconnect
            return 0;
        }
        else
        {
            // Received a random UA. Let's just ignore
            continue;
        }
    }

    // Read successfully the correct package
    // Destuff to a buffer with one extra space for the bcc character
    res = destuff(&(link->frame), bufferWithBcc);
```

```
        if (res == -1)
        {
            // Wrong character found after escape character
            corrupted = 1;
        }
        else //Destuffing was ok
        {
            // Verify protection byte
            verifyBcc = verifyProtectionByte(bufferWithBcc, res);
        }

        if (!verifyBcc || corrupted)
        {
            // Was already corrupted or the bcc was wrong
            corrupted = 1;
        }
        else // Protection byte was ok
        {
            // Copy to the buffer return buffer
            res -= 1; // Res included the protection byte. The message is in fact one byte
shorter
            memcpy(buffer, bufferWithBcc, res);
        }
        if (corrupted)
        {
            // There was corruption in the data
            if (Nr != link->sequenceNumber)
            {
                // Corrupted but it was from a previous package.
                // Ask for current package
                RRMessage(link->frame.frame, link->sequenceNumber);
                link->frame.frameUsedSize = CMDSZ;
                writeLinkResponse(link);
            }
            else
            {
                // The current package was corrupted
                REJMessage(link->frame.frame, link->sequenceNumber);
                link->frame.frameUsedSize = CMDSZ;
                writeLinkResponse(link);
            }
```

```
            // Go back to reading
            continue;
        }
        else
        {
            // Ask for next package
            link->sequenceNumber = (link->sequenceNumber + 1) % 2;
            RRMessage(link->frame.frame, link->sequenceNumber);
            link->frame.frameUsedSize = CMDSZ;
            writeLinkResponse(link);
        }
        return res;
    }
}
```

Annex 40 (setFD.h)

```
#ifndef SETFD_H
#define SETFD_H

#include <termios.h>
#include "dataProtocol.h"

// Change according to Baudrate to be used
// #define BAUDRATE B4800
// #define BAUDRATE B9600
// #define BAUDRATE B19200
#define BAUDRATE B38400
// #define BAUDRATE B57600
#define _POSIX_SOURCE 1 /* POSIX compliant source */
#define FALSE 0
#define TRUE 1

int setFD(struct linkLayer *link, struct termios *oldtio, struct termios *newtio);

int closeFD(int fd, struct termios oldtio);

#endif //SETFD_H
```

Annex 41 (setFD.c)

```c
#include "setFD.h"

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

int setFD(struct linkLayer *link, struct termios *oldtio, struct termios *newtio)
{

    int fd;
    /*
    Open serial port device for reading and writing and not as controlling tty
    because we don't want to get killed if linenoise sends CTRL-C.
  */

    fd = open(link->port, O_RDWR | O_NOCTTY);

    // Makes the fd nonblocking. We'd rather achieve that by another method since this will
make
    // the read to throw error 11: Resource temporarily unavailable
    // int flags = fcntl(fd, F_GETFL, 0); fcntl(fd, F_SETFL, flags | O_NONBLOCK);

    if (fd < 0)
    {
        perror(link->port);
        exit(1);
    }

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

    bzero(newtio, sizeof((*newtio)));
```

```
    (*newtio).c_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;
    (*newtio).c_iflag = IGNPAR;
    (*newtio).c_oflag = 0;

    /* set input mode (non-canonical, no echo,...) */
    (*newtio).c_lflag = 0;

    //Currently, unblocks read after timeout with 0.1 seconds
    (*newtio).c_cc[VTIME] = 1; /* inter-character timer unused */
    (*newtio).c_cc[VMIN] = 0;  /* blocking read until 5 chars received */

    /*
    VTIME e VMIN devem ser alterados de forma a proteger com um temporizador a
    leitura do(s) pr?ximo(s) caracter(es)
  */

    tcflush(fd, TCIOFLUSH);

    if (tcsetattr(fd, TCSANOW, newtio) == -1)
    {
        perror("tcsetattr");
        exit(1);
    }

    link->fd = fd;
    return 0;
}

int closeFD(int fd, struct termios oldtio)
{
    if (tcsetattr(fd, TCSANOW, &oldtio) == -1)
    {
        perror("tcsetattr");
        exit(1);
    }

    close(fd);
    return 0;
}
```

Annex 42 (transmitter.h)

```
#ifndef TRANSMITTER_H
#define TRANSMITTER_H

#include "dataProtocol.h"

int openTransmitter(struct linkLayer *link);

int writeTransmitter(struct linkLayer *link, u_int8_t *buffer, int length);

int closeTransmitter(struct linkLayer *link);

#endif //TRANSMITTER_H
```

Annex 43 (transmitter.c)

```
#include "transmitter.h"

#include "commandMessages.h"
#include "physicalProtocol.h"
#include "byteStuffing.h"
#include "dataProtection.h"

int openTransmitter(struct linkLayer *link)
{

    SETMessage(link->frame.frame);
    link->frame.frameUsedSize = CMDSZ;
    return writeLinkCommand(link, A_EM, UA);
}

int writeTransmitter(struct linkLayer *link, u_int8_t *buffer, int length)
{
    int res;
    u_int8_t protectionByte;
    IMessage(link->frame.frame, link->sequenceNumber);
    protectionByte = createProtectionByte(buffer,length);
    link->frame.frameUsedSize = stuff(&(link->frame), buffer, length, protectionByte);
    res = writeLinkInformation(link, A_EM);
    if (res == 0)
    {
```

```
        link->sequenceNumber = (link->sequenceNumber + 1) % 2; // Changes the sequence
number. N = 2
        return length; // Return number of bytes transmitted
    }
    return -1;
}

int closeTransmitter(struct linkLayer *link)
{
    int res = 0;
    u_int8_t mes[6];

    DISCMessage(link->frame.frame, A_EM);
    link->frame.frameUsedSize = CMDSZ;

    if (writeLinkCommand(link, A_REC, DISC))
    {
        //Timeout receiving disc
        return -1;
    }

    UAMessage(link->frame.frame, A_REC);
    link->frame.frameUsedSize = CMDSZ;
    writeLinkResponse(link);

    return 0;
}
```