# RCOM - Project 1 - Data link protocol

Bernardo Manuel Esteves dos Santos up201706534
David Luís Dias da Silva up201705373
Luís Pedro Pereira Lopes Mascarenhas Cunha up201706736

## Summary

Computer networks are frequently described in a layered model. The data link layer regulates data flow, tries to eliminate transmission errors and provides an interface of services to it's upper layer(Network Layer). This project aims to develop a data link protocol and an application that makes use of said protocol to transfer files between two computers using a serial port connection.

We were able to produce a program that implemented both protocols and that was resistant to errors (disconnecting and reconnecting serial port mid-transmission and introducing electric interference on the connection). The achieved efficiencies behaved as expected according to the parameters, however, due to empirical factors, were not equal to the theoretical values.

## 1. Introduction

The goal of the project is to apply the knowledge of computer network organization and concepts acquired during the lectures, by implementing a data link layer protocol capable of transmitting data between two computers using a serial port connection. An application was developed to use the data-link protocol coupled with an application level TLV-based protocol to transfer a file between two computers connected using the serial port.

In the following sections the major points of the application that was developed will be described in detail. Section 2 describes the architecture(functional blocks and interfaces) of the system. The following section describes the code structure with the first subsection specifying the API of each layer(link and application) and what functions and data structures have a major role in said API. Section 3.2 describes the main use-cases and program-flow. Sections 4.1 and 4.2 describe, respectively, the data-link and application protocol that were used and their implementation details. The last two sections aim at discussing the development process of the program (validation and testing) and an empirical evaluation of the data-link protocol.

## 2. Architecture

The architecture of the system closely follows the layered organization that is common in networks. The system is comprised of two main blocks, the application block and the data link block.

The latter has the lowest level of abstraction of the two and its main function is to write/read bytes from the serial port. Because this protocol happens at the data link layer, it also has the responsibility of reducing/eliminating the transmission errors that might happen on the transfer, through techniques such as ARQ(Automatic Repeat Request) or a BCC(Block check character) that will be described in detail in section 4.1. This block's interface contains functions that open/close the communication(*llopen*, *llcose*) and read/write bytes from the serial port (*llread*, *llwrite*).

The application block has the purpose of transferring a file (given through an argument, among other configuration settings) to another computer that is connected to it via de serial port. This block handles the argument parsing and the mechanisms that, depending on the role of the system in the transfer, split or unite the file into/from chunks. These chunks are traded using a TLV-based protocol (described in section 4.2). The provided interface consists of two functions, one for receiving (*receive_file*) and one for sending (*send_file*) a file.

To follow the layered model, the data link block only makes use of functions from a lower level(layer) of abstraction, in this case, the kernel calls for writing/reading to/from the serial port, and the application makes use of the interface provided by the data link layer to transfer the file.

# 3. Code structure

## 3.1 API, internal functions and data structures

The project offers two API's, one for each conceptual block described in the previous section. Both API's only include a small set of functions, two for the application API and four for the data link layer API.

### 3.1.1 Data link layer

The link layer interface is comprised of four functions. The llopen function (*int llopen(int port, int role)*) receives the number of the port which will be opened for transfering data and the caller's role (0 for receiver and 1 for transmitter, see macros *RECEIVER* and *TRANSMITTER*), opens the corresponding serial-port connection ( /dev/ttySX, where X is the value of port) and, depending on the role, sends the appropriate command or response to the partner according to the data link layer protocol(see section 4.1). This function is a blocking call and will wait for the other party to open the same port with the opposite role. Similarly, the llclose function(*int llclose(int fd)*) sends the needed messages to close the communications. The llwrite function (*int llwrite(int fd, uint8_t * buffer, int length)*) sends *buffer* to the receiver through the port whose file descriptor is *fd*, returning the number of bytes written or -1 when an error occurs. The last function llread (*int llread(int fd, uint8_t * buffer)*) reads the available bytes from the serial port corresponding to fd into *buffer* and returns the number of bytes read. The function llwrite isn't a blocking call and will return with error after a certain number(configurable by the macro MAX_TIMEOUT_RETRY_CNT) of failed (by timeout) writing attempts. llread is a blocking call and , as such, won't return until some bytes have been read or the writer issues a disconnecting call (llclose) in which the function will return 0.

The data link layer is configured by the following set of macros: BAUDRATE, MAX_TIMEOUT_RETRY_CNT, VTIME_VALUE, VMIN_VALUE.

The data link layer is composed of two main modules that implement different aspects of the protocol. The "main" one, is the module that controls the logic, kernel calls and data-flow necessary to implement the protocol. For example functions like *serial_port_setup*, *alarm_handler*, issue calls that open the serial port files or define the handler for the timeout alarms. Functions such as *llread*, *llwrite*, *write_data*, *write_frame, sendAck, sendUA, process_read_i_frame, process_read_su_frame, receiver_open/close, transmitter_open/close*, implement the logic needed to assure compliance with the protocol and issue the correct kernel *read/write* calls to the serial port. There are also auxiliary functions needed to build frames and validate data BCCs, such *build_su_frame* and *valid_data_bcc.* The second component of this module are the state machines (*receiver_state_machine.h* and *transmitter_state_machine.h*) that are used to interpret the bytes that are received (one-by-one) from the communication partner. The state machine has, as indicated by its name, a current state that changes according to the byte that is read (*void tsm_process_input(struct transmitter_state_machine* st_machine, uint8_t received_byte)*) taking the necessary actions such as building the frame, checking BCC values, addresses, commands and destuffing (see section 4.1 for more details). There are two data structures that support this module which are the *linklayer_info* struct and the state machines. The first one holds information about the baud rate, role (receiver or transmitter), sequence number of packets and the receiver state machine. The state machines, as is said before, hold information about the reading status, connection, allocated memory and read data.

### 3.1.2 Application layer

The application layer is comprised of two functions, one that sends a file and one that receives a file.

The send_file function (*int send_file(int port, char* file_path, size_t packet_size)*) receives the number of the port that will be open to transfer the file located at *file_path* and also receives the maximum size (in bytes) for the packets. All the data that is going to be transferred is stored in packets, which are simply arrays of bytes. Two types of packets can be assembled: data packets(*uint8_t* build_data_packet(uint8_t* data, size_t data_size)*) and control packets (*uint8_t* build_control_packet(packet_type type, int* packet_size, tlv* tlv_list[], const uint8_t tlv_list_size)*), the *build_control_packet* function receives information about the file in a list of *tlv* objects, each contains the type of the data that is stored(file name or size), an array of bytes containing the value, the length of said array and data type of the value(string or integer). These objects can be created using one of two functions depending if the value that will be stored is a string (tlv* create_tlv_str(data_type type, char* value)) or an integer(*tlv* create_tlv_int(data_type type, int value)*).

The receive_file function (*int receive_file(int port)*) receives the number of the port that will be opened to receive a file.

In order to test the application layer we created a program that uses its API. We have made a makefile to make the testing easier, the binary has the name "penguin". On the computer that is sending the file the program should be run as follows:

./penguin transmitter [port] [path] [packet_size]

On the receiving end:
./penguin receiver [port]

## 3.2 Main use cases

### 3.2.1 Data link layer

The main use case for the data link module can be simplified as sending an array of bytes from one computer(A, transmitter) to another(B, receiver). Sending a more complex structure may involve multiple calls of the reading and writing functions. To start, each computer calls *llopen* with their respective role, which sets up the serial port settings (*serial_port_setup*) and calls a function depending on the role. Computer A's call to *llopen* will call the *transmitter_open* function which builds the needed frame with *build_su_frame* and calls *write_frame* which sends the built frame and waits for a response. Computer B's *llopen* calls the *receiver_open* function that waits for a connecting call from computer A and sends the appropriate response with the call to *process_read_su_frame* and, subsequently, *sendUA*.

Computer A calls *llwrite,* that calls the *write_frame* function , which calls the *write_data* function and waits for the response from computer B, writing the data again if need accordingly to the response. The *write_data* function is responsible for writing each byte of the array to the opened port, preceded by the frame header, with the appropriate stuffing and followed by the data BCC (which it calculates). Simultaneously, computer B calls *llread* that waits for data from A and processes it accordingly with *process_read_i_frame*. This function validates the read frame and issues the appropriate response to A through *sendAck* (ACK or NACK, depending on the error occurred, such as invalid data BCC or wrong sequence number).

When both *llread* and *llwrite* return, both computers call *llclose*. Depending on the role the function calls *transmitter/receiver_close* which send messages and wait for responses (according to the protocol) with the usage of *build_su_frame, sendAck, sendUA* and *process_read_su_frame*. llclose ends by reverting the serial port's settings.

Throughout the mentioned functions, whenever a byte is read from the serial port, the functions *tsm_process_input* and *sm_process_input* are called. These functions change the states of the state-machines and build the read frames. The state machines are created in *transmitter/receiver_open* and are destroyed in *llclose*.

### 3.2.2 Application layer

The main use case for the application module is sending a file from one computer (A, sender) to another (B, receiver) that are connected via serial port.

To start the transfer computer A must provide its role, the number of the port that will be used for communication, the path to the desired file and the maximum size of the packets.

Firstly, the sender will open the file and get all the information needed to build the start control packet(file name and size), a list of *tlv* objects is created and the control packet is built. After this the communication with the receiver will be initiated by calling *llopen*. The first packet to be sent is the start control packet, after which the sender starts reading the file, chunk by chunk, building the data packets with the information that is being retrieved and sending them to the receiver by writing the packet to the serial port (*llwrite*). At the end of the file, an end control packet is assembled and sent, the file is closed and the communication is terminated by calling *llclose*.

Computer B only has to provide its role and the number of the serial port. It will call *llopen* with the given port number and if a connection is established it will begin to read from the port by calling *llread*, the first packet read must be a start control packet or else the application will exit in error status. After reading the start control packet its information is stored in a *control_info* object and a file with the name provided in the packet is created. The receiver will now read packets from the serial port always checking if they are data packets, in which case they will be interpreted and their data will be written to the file that was created, or if the packet that was read is an end control packet, which means that the file transfer is complete. When the transfer is complete the program stops reading from the port and extracts the information from the end packet to a *control_info* object in order to compare it to the one from the beginning and determine if there were any errors. At last, the file is closed and the communication between the computers is terminated with a call to *llclose.*

# 4. Protocols

## 4.1 Data link protocol

The data link layer protocol is based on frames that can be of three types: information(I), supervision(S) and unnumbered(U). The frames are delimited by a eight bit flag (0x7E), whose transparency is assured by the usage of byte stuffing. The types of frames have a common header(flag, address, control, BCC) but only type I frames carry a data segment. Every frame is protected by a BCC(Block Check Character), however, the protection is simple in S/U frames and double for the I frames, where there are two independent BCC fields for the header and data segment. The protocol also controls errors through a Stop and Wait ARQ mechanism. When a frame is sent(with a given sequence number), the transmitter waits for a response from the receiver. Depending on the response, the transmitter may send the next frame or retransmit the previous one. The connection is started by sending a S/U frame with a start command(and appropriate response), then both parties trade I frames (data) and S/U response frames(acknowledgements) and the connections ends with a S/U disconnect command (and appropriate response).

An example for the usual implementation for sending a frame (data or command) and waiting for a response can be seen in code segment 1. An outer while-loop makes the function try to write the data again after a timeout for three times. The inner loop reads a byte from the serial port and processes it while a timeout doesn't occur. In each iteration of the inner loop a byte is processed by a state machine. Receiving a frame was implemented in a similar way, except there is only one loop and the function will run until an actual frame is read. An example of the stop and wait mechanism can be seen in the inner loop where if a program receives a NACK it will try to send the same frame again.

Segment 5 shows the process of sending a frame (in this case a type I frame), calculating the stuffing for its data segment and calculating the check characters.

An example of how the state machine processes a byte and updates its state can be scene in code segment 2, that represents two of the state machines transition handlers. This code segment also shows how the byte de-stuffing is done, where if a an escape code is found, the state machine transitions to a "temporary state" and if the correct follow up code is received in the next byte the appropriate byte is added to the result array.

The header BCC is checked during frame processing inside the state-machine (code segment 3) and the data BCC is checked in the *process_read_i_frame* function (code segment 4).

Segment 6 shows the process of writing an ACK (which is very similar to the function for writing a unnumbered acknowledgement).

## 4.2. Application protocol

The application layer protocol can is based on packets. Firstly the transmitter must collect the necessary information (file size and name) and enclose it in a start control packet so it can be transferred to the receiver, after this data packets containing the contents of the file are sent to the receiver until the end of the file is reached. At this point, an end control packet is built and sent to indicate the end of the file transfer. On the receiver side, the start packet is received, stored and its contents used to create a new file to which the data packet contents will be written. When the end control packet is received the information extracted from both the control packets is compared to ensure that no errors occurred.

Data packets are composed of a control field with a value of 1 (indicating that this is a data packet), the sequence number (module of 256), two fields that indicate the length of the data, the first (L1) is the integer division of the length by 256 and the second (L2) is the remainder of this division, the rest of the packet is L bytes of data (L = 256 * L2 + L1).

Control packets have a completely different structure. The control field can have two values: 2, if it is a start packet, or 3, if it is an end packet. Each following parameter is stored in the TLV(Type, Length, Value) format: type is one byte long and indicates what is the parameter(0 - file size, 1 - file name), length is also one byte long and indicates the size of the value field, and value contains the value of the parameter.

# 5. Validation

To test both API's we ran a series of tests. First, for the link layer API we sent several single frames and sequences of frames and saw if the interpreted those frames correctly and, also, if the transmitter parsed the responses correctly. This was made by the use of logging messages which described in detail the actions that were made by the program and the sequences of transitions that happened in the state machine. To test the correctness of the protocol we sent frames that contained errors in the data and checked if the receiver detected them, and issued the correct response (sending a NACK) to which we tested if the transmitter retransmitted the appropriate frame. Also we sent data segments that contained different sequences of the escape and flag bytes to validade the byte stuffing and destuffing. Throughout this process the state machine was checked for the correct transitions. The API passed all tests and followed the proposed protocol. On a larger scale, through the tests of the application API described ahead, we tested the behaviour of the link layer when transmitting a large set of frames (of a large file) and by causing interference with the serial port connection. Even with large amounts of interference, the protocol delivered the expected data in the expected order with no loss. Different settings of the link layer like the baud rate were also changed in order to test it.

The application API was tested by sending files between two computers. We experimented with sending different packet and file sizes and different file types. The program behaved as expected and all files were sent without an loss.

## 6. Data link protocol efficiency

To test the implemented ARQ Stop & Wait protocol, we did a set of execution time measures of our program, calculated the efficiency, and compared the results to the expected theoretical value, calculated using:

Efficiency: S = 1 / (1 + 2*a)

a = Propagation time / Frame transmission time

In order to calculate the propagation time, we used the following values:

Cable length: d = 5m

Propagation time: 5 µs/km

We measured the time in between before the first and last I frames were sent in our file transfer application. We performed the following tests, in each varying a different variable.

### 6.1 Varying size of I frames

In this test, we sent a file with size 49152 bytes, with a baudrate of 38400 bit/s. With the results of this test, we concluded that the larger the size of the I frame, the higher the efficiency of the protocol. On the plot (section 8.2.1) we can observe that for low packet sizes the efficiency starts at around 56% and stabilizes at around 78%.

### 6.2 Varying baudrate

In this test, we sent a file with size 49152 bytes, several times, each time increasing the baud rate (from 4800 bits/s to 115200 bits/s). As expected, the total time taken to transmit decreased with an inverse proportion as the baud rate increased. The efficiency was stable (≈ 78%). (Values and plot in section 8.2.2)

### 6.3 Varying propagation time

The propagation time was simulated by introducing a delay in the packet sending (the actual cable was about 5m). As the propagation time increased the efficiency decreased linearly starting at about 78%. (Values and plot in section 8.2.3)

### 6.4 Varying frame error rate 13200

To test how our protocol behaved with different frame error rates, we simulated errors by calculating the wrong BCC1 and BCC2, independently, in I-frames. In all time measures we used a file with size 13200 bytes, sent in 100 packets of 132 bytes each, using a baud rate of 115200 bit/s.

When introducing errors in BCC1 (header errors), the efficiency decreased drastically, because the receiver discarded the bytes read until the error which caused the transmitter to resend the same frame after it timed out (3 seconds).

When introducing errors in BCC2 (frame body errors), the efficiency decreased linearly, as expected. This is due to the fact that the receiver, when encountering a BCC2 error, sends a NACK, which causes the transmitter to immediately retransmit the frame.

(Values and plot in section 8.2.4)

## 7. Conclusions

The layered model common in networks creates an environment propitious to reuse due to its modularity and allows for the development of more complex systems since a higher level API must only know the the interface of the layer below. The stop and wait ARQ mechanism is a very simple and robust mechanism, however these advantages come at a cost, since the efficiency suffers with increasing distance and file size. The BCC mechanisms used, although simple, also proved useful in the error detection, however it’s simplicity may prove a fault in error-prone connections since the “size” of the error that it can detect is small.

Although the empirical calculated efficiencies behaved as expected according to the tested parameters, the actual values diverged from the theoretical ones.

## 8. Attachments

### 8.1 Code Examples

#### 8.1.1 Segment 1 - in function write_frame from linklayer.c (simplified)

```
// Try to send the frame (if the sending failed 3 times, return with error)
  while (numTries <= MAX_TIMEOUT_RETRY_CNT)
  {
    ...

    // Send data or S/U frame
    if (type == DATA)
      n_written = write_data(fd, buffer, length);
    else
      n_written = write(fd, buffer, SU_FRAME_SIZE);
    ...

    // Wait for answer from the receiver (until timeout happens)
    while (!timedOut)
    {
      // Read a byte from the serial port
      res = read(fd, &currentByte, 1);

      // State-machine processes the read byte
      tsm_process_input(&st_machine, currentByte);

      if (st_machine.currentState == T_STATE_STOP)
      { // SU frame reading stop-state reached

        if (type == DATA)
        {

          if (st_machine.frame[2] == ((nextSequenceNumber << 7) | CONTROL_RR_BASE))
          {
            // Expected ACK received (with the next sequence number)
            // Update the sequence number (if 1 -> 0, if 0 -> 1)

            connection_info.sequenceNumber = nextSequenceNumber;
            return n_written;
          }
          else if (st_machine.frame[2] == ((connection_info.sequenceNumber << 7) |
CONTROL_REJ_BASE))
          {
            // NACK received, try to send the same data frame again
            // numTries is reseted because no timeout occured (a transmission error
happened)

            numTries = 1;
            break;
          }
        }
        ...
      }
    }
  }
```

### 8.1.2 Segment 2 - functions sm_i_data_rcv_st_handler and sm_esc_found_st_handler from receiver_state_machine.c (simplified)

```
void sm_i_data_rcv_st_handler(struct receiver_state_machine *st_machine, uint8_t
receivedByte)
{

  if(st_machine->currentByte_idx == st_machine->allocatedMemory){

    st_machine->allocatedMemory *= 2;
    st_machine->frame = realloc(st_machine->frame, st_machine->allocatedMemory);
  }

  if (receivedByte == 0x7d)
  { // escape character must be converted to original chararacter
    st_machine->currentState = R_STATE_ESCAPE_FOUND;
  }
  else if (receivedByte == FLAG)
  { // end of the frame. This means the previous byte was the BCC of the data and must
be checked

    if(st_machine->currentByte_idx < 6){     // the end flag was received but no bcc and
or data was received

      st_machine->currentState = R_STATE_START;
      return;
    }

    st_machine->frame[st_machine->currentByte_idx] = receivedByte;
    st_machine->currentByte_idx++;
    st_machine->currentState = R_STATE_I_STOP;
  }
  else
  {
    st_machine->frame[st_machine->currentByte_idx] = receivedByte; // normal data byte
    st_machine->currentByte_idx++;   }
  return;
}

void sm_esc_found_st_handler(struct receiver_state_machine *st_machine, uint8_t
receivedByte)
{


  if (receivedByte == 0x5d)
  {

    st_machine->frame[st_machine->currentByte_idx] = 0x7d; // parse escape byte
    st_machine->currentByte_idx++;
    st_machine->currentState = R_STATE_I_DATA_RCV;

  }
  else if (receivedByte == 0x5e)
  {
    st_machine->frame[st_machine->currentByte_idx] = 0x7e; // parse flag byte
    st_machine->currentByte_idx++;
    st_machine->currentState = R_STATE_I_DATA_RCV;
  }
  else
  {
    st_machine->currentState = R_STATE_START;
  }

  return;
}
```

### 8.1.3 Segment 3 - sm_su_c_rcv_st_handler function at receiver_state_machine.c

```c
void sm_su_c_rcv_st_handler(struct receiver_state_machine *st_machine, uint8_t
receivedByte)
{

  if (receivedByte == (st_machine->frame[CTRL_INDEX] ^ st_machine->frame[ADDR_INDEX]))
  {
    st_machine->currentState = R_STATE_SU_BCC1_OK;
    st_machine->frame[BCC1_INDEX] = receivedByte;
  }
  else if (receivedByte == FLAG)
    st_machine->currentState = R_STATE_FLAG_RCV;
  else
    st_machine->currentState = R_STATE_START;

  return;
}
```

### 8.1.4 Segment 4 - valid_data_bcc function at linklayer.c

```c
bool valid_data_bcc(uint8_t frame[], size_t frame_size)
{

  const int BCC2_INDEX = frame_size - 2;
  const int I_FRAME_DATA_END_INDEX = frame_size - 3;

  uint8_t bcc_value = frame[BCC2_INDEX];
  uint8_t calculated_value = 0;

  for (int i = I_FRAME_DATA_START_INDEX; i <= I_FRAME_DATA_END_INDEX; i++)
    calculated_value ^= frame[i];

  return bcc_value == calculated_value;
}
```

### 8.1.5 Segment 5 - write_data function at linklayer.c (simplified)

```c
int write_data(int fd, uint8_t *buffer, int length)
{

  ...
  // Build the data frame header
  uint8_t frame[I_FRAME_HEADER_SIZE];
  frame[FLAG_START_INDEX] = FLAG;
  frame[ADDR_INDEX] = ADDR_TRANSM_COMMAND;
  frame[CTRL_INDEX] = connection_info.sequenceNumber ? 0x40 : 0x00;
  frame[BCC1_INDEX] = frame[1] ^ frame[2];

  // Write header to serial port
  write(fd, frame, I_FRAME_HEADER_SIZE);

  ...

  uint8_t bcc2 = 0x00;

  // Data writting to serial port (byte by byte)
  for (int i = 0; i < length; i++)
  {

    // Update BCC2
    bcc2 ^= buffer[i];

    if (buffer[i] == FLAG || buffer[i] == ESC)
    {

      // Handling byte stuffing
      frame[0] = ESC;
      frame[1] = buffer[i] ^ ESC_XOR;
      nWritten = write(fd, frame, 2);

      ...
    }
    else
    {
      nWritten = write(fd, &buffer[i], 1);
      ...
    }
  }

  // Write the Data bcc

  if (bcc2 == FLAG || bcc2 == ESC)
  { // Stuff the data bcc if necessary
    frame[0] = ESC;
    frame[1] = bcc2 ^ ESC_XOR;
    nWritten = write(fd, frame, 2);
    ...
  }
  else
  {
    nWritten = write(fd, &bcc2, 1);
    ...
  }

  // Write the final flag
  frame[0] = FLAG;
  nWritten = write(fd, frame, 1);
  ...
}
```

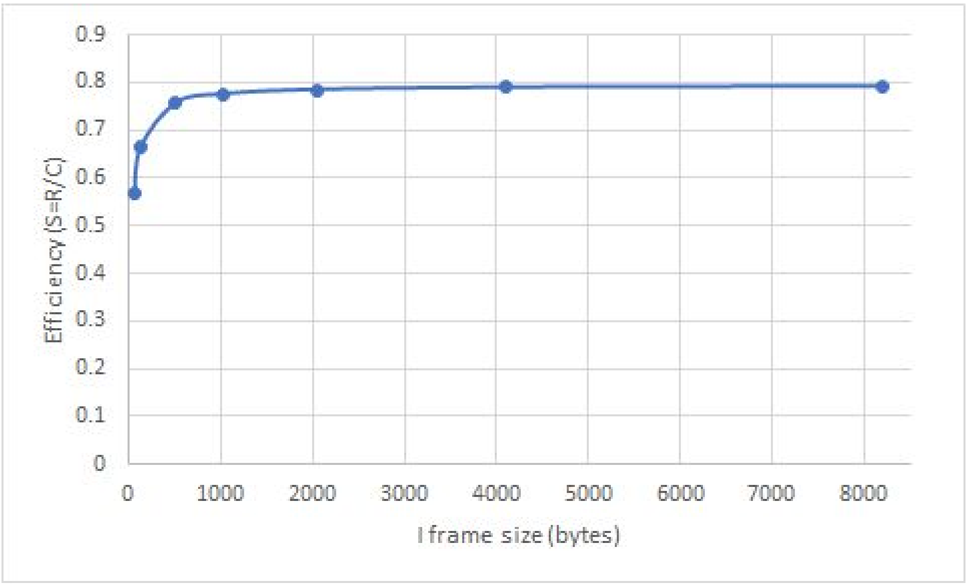
### 8.1.6 Segment 6 - sendAck function at linklayer.c (simplified)

```
int sendAck(int fd, bool type, int sequence_no)
{
  // Build the control byte of the ACK frame
  uint8_t sequence_byte = sequence_no ? BIT(7) : 0x00;
  uint8_t control_byte = type ? (CONTROL_RR_BASE | sequence_byte) : (CONTROL_REJ_BASE |
sequence_byte);
  uint8_t response[SU_FRAME_SIZE];

  build_su_frame(response, ADDR_RECEIV_RES, control_byte);
  int res = write(fd, response, SU_FRAME_SIZE);

  ...
  return 0;
}
```

## 8.2 Efficiency calculations

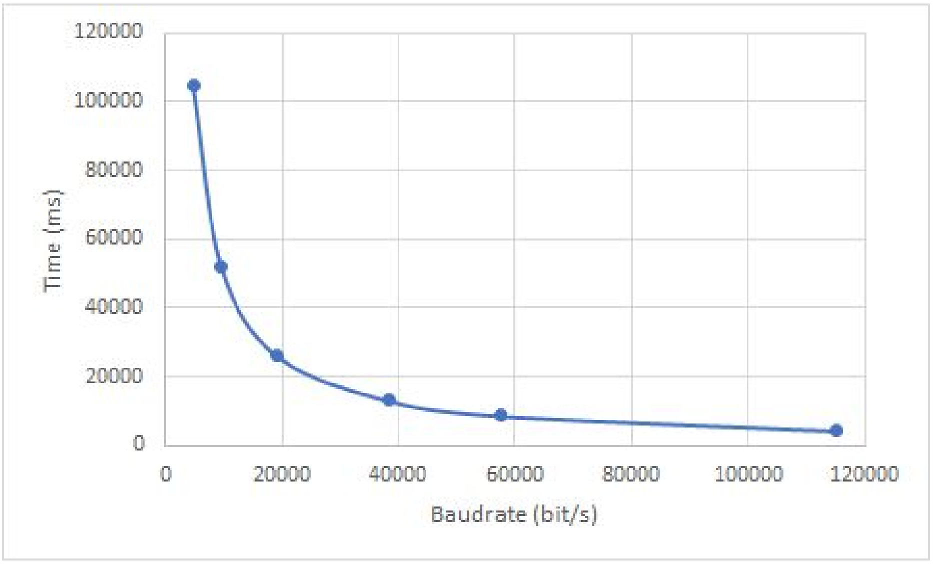
### 8.2.1 Calculation 1 - Varying size of I frames

| I frame size (bytes) | Time (ms) | Efficiency (S = R/C) | Theoretical Efficiency |
|---|---|---|---|
| 64 | 17966 | 0.56997 | 0.99256 |
| 128 | 15424 | 0.66391 | 0.99626 |
| 512 | 13521 | 0.75734 | 0.99906 |
| 1024 | 13201 | 0.77571 | 0.99953 |
| 2048 | 13048 | 0.78482 | 0.99977 |
| 4096 | 12970 | 0.78952 | 0.99988 |
| 8192 | 12929 | 0.79202 | 0.99994 |



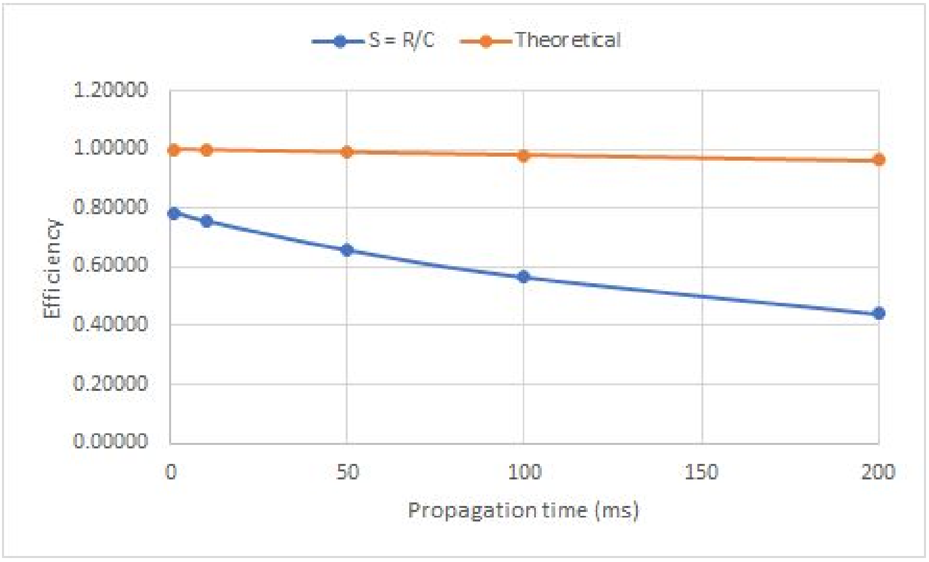
### 8.2.2 Calculation 2 - Varying baudrate

| Baudrate (bit/s) | Time (ms) | Efficency (S = R/C) | Theoretical Efficiency |
|---|---|---|---|
| 4800 | 104601 | 0.78317 | 1.00000 |
| 9600 | 52309 | 0.78304 | 1.00000 |
| 19200 | 26162 | 0.78281 | 1.00000 |
| 38400 | 13086 | 0.78249 | 0.99999 |
| 57600 | 8727 | 0.78226 | 0.99999 |
| 115200 | 4368 | 0.78140 | 0.99997 |



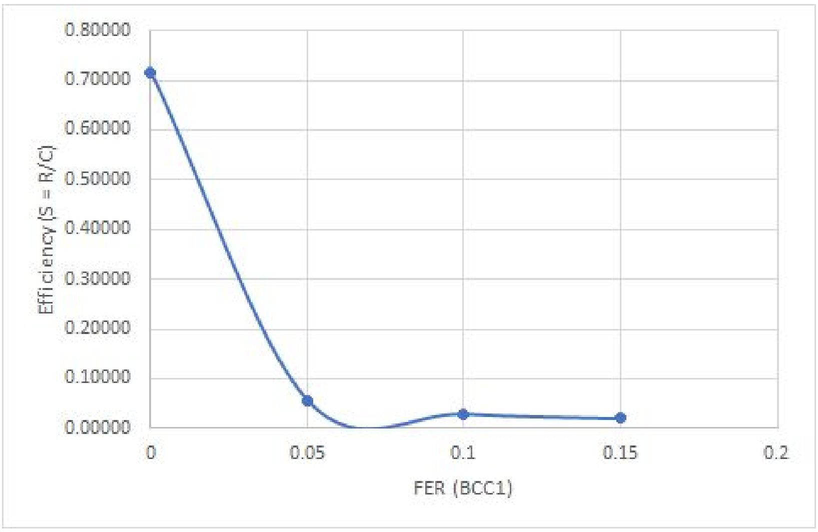
## 8.2.3 Calculation 3 - Varying propagation time

| Propagation Time (ms) | Time (ms) | Efficency (S = R/C) | Theoretical Efficiency |
|---:|---:|---:|---:|
| 1 | 13081 | 0.78281 | 0.99980 |
| 10 | 13591 | 0.75343 | 0.99805 |
| 50 | 15619 | 0.65561 | 0.99033 |
| 100 | 18156 | 0.56401 | 0.98084 |
| 200 | 23306 | 0.43937 | 0.96241 |

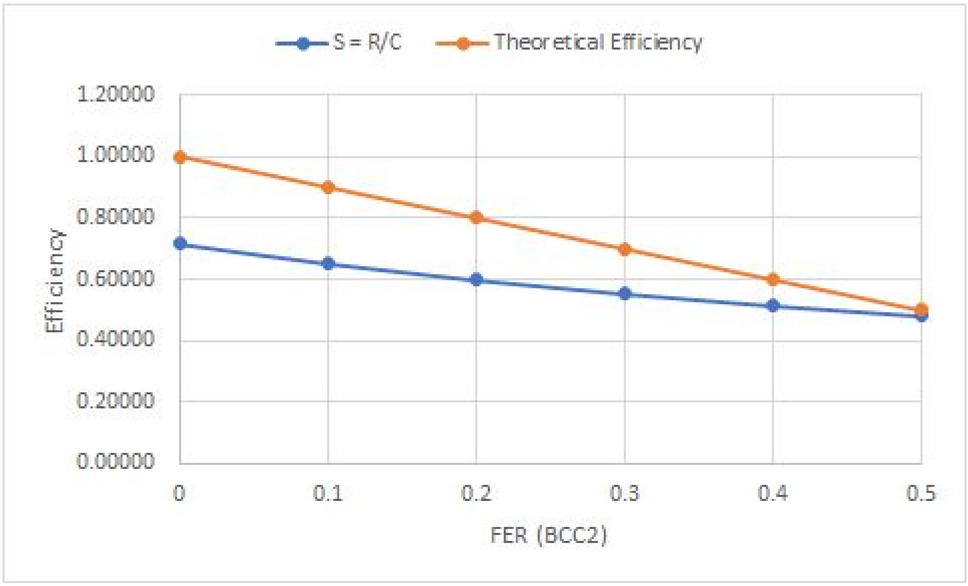


### 8.2.4 Calculation 4 - Varying frame error rate

| FER (BCC1) | Time (ms) | Efficiency (S = R/C) |
|---|---|---|
| 0 | 1279 | 0.71671 |
| 0.05 | 16279 | 0.05631 |
| 0.1 | 31282 | 0.02930 |
| 0.15 | 46283 | 0.01981 |



| FER (BCC2) | Time (ms) | Efficiency (S = R/C) | Theoretical Efficiency |
|---|---|---|---|
| 0 | 1279 | 0.71671 | 0.99989 |
| 0.1 | 1405 | 0.65243 | 0.89990 |
| 0.2 | 1532 | 0.59835 | 0.79991 |
| 0.3 | 1657 | 0.55321 | 0.69992 |
| 0.4 | 1782 | 0.51440 | 0.59993 |
| 0.5 | 1908 | 0.48043 | 0.49995 |


S = R/C
Theoretical Efficiency
Efficiency
FER (BCC2)