RC - Serial Port Project

by José Castro and Pedro Silva - T14G9

Faculty of Engineering of the University of Porto

1. Introduction

The goal of this project is to develop a digital infrastructure to serve as the basis for information exchange through a serial port. This infrastructure is divided into two layers - the application layer and the data link layer - and each of these operate independently of each other.

This report will serve as a written specification of what was achieved in this development process.

- Architecture

- Code Structure

- Main Use Cases

- Data Link Protocol

- Application Protocol

- Validation

- Data Link Protocol Efficiency

- Conclusions

2. Architecture

2.1 - Data Link Layer

The Data Link Layer is the lowest level of the application and is responsible for supervising direct communication with the serial port, detecting errors and handling them.

2.2 - Application Layer

The application layer is above the link layer in terms of leveling and is responsible for handling file contents and sending them through the link layer. Information is retrieved from a file and sent through the link layer, which is then received in this same link layer: the application layer simply takes care of unpacking it in the receiver's end.

2.3 - Main

Simple interface that runs transmitter or receiver code depending on the role given during the program execution:

serialport role [filepath]

where:

- *serialport* is the port's path (e.g.: dev/ttyS01)

- *role* is the role *({rx, tx}* where *rx* is the receiver, *tx* is the transmitter)

- *[filepath]* is the file-to-send's path (only needed if initializing the program as transmitter)

3. Code Structure

3.1 - Application Layer

The application layer stores every function and usable without any regard for who is using it, be it is the receiver or the transmitter. Important parameters for the application layer to know about would be the file descriptor and the role it is currently executing, however, these are both specified in *main.c* and, as such, there is no need for them.

application_layer.h & *application_layer.c*

```
typedef struct {
    char* filename;
    u_int8_t filename_size;
    long filesize;
} fileStruct;
```

A struct is used to store the file we're sending and to make a new name in the receiver end.

The *filename* field is a character buffer that contains the file's (relative) path.

The *filename_size* field contains the size of the *filename* buffer.

The *filesize* field contains the size of the file.

```c
typedef struct {
    unsigned char* packet;
    int packet_size;
} startPacket;
```

A struct is used to store the starting control packet sent to then send it again as an end packet at the end of the program's data transfer.

The *packet* buffer stores the packet's contents.

The *packet_size* specifies the *packet* buffer size.

```c
typedef struct {
    u_int8_t T;
    u_int8_t L;
    u_int8_t* V;
} TLV;
```

A struct is used to store the TLV (Type, Length, Value) section of a control packet.

The *T* field represents the Type parameter (0 – file size, 1 – file name).

The *L* field represents the Length parameter (the size of the V field in octets).

The *V* field contains the data to be sent (file size or file name).

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

Effectively starts and establishes the connection between transmitter and receiver and defines the roles in the current context. Uses link layer's *llopen*

```c
int readFile();
```

Reads the file sent through the serial port. It gets its general information first by calling *readControlPacket*, followed by the file's contents by calling *writeFileContents*.

```c
int readControlPacket();
```

Reads the file's general information. Calls *readTLV* for each group of parameters in the TLV format, with the index at the start of the next group and a reference to the *tlv* struct.

```c
int readTLV(unsigned char *packet, TLV
*tlv);
```

Reads one group of parameters in the TLV format, given by *packet*. Each parameter is stored in the appropriate variable of the *tlv* struct.

```c
int writeFileContents(FILE *fp);
```

Calls *llread* to read the file's contents, and writes them to *fp*.

```c
int sendFile(char* path);
```

Sends the file, obtained by following *path*, through the serial port. It starts with its general information first by calling *sendControlPacket*, followed by the file's contents by calling *sendFileContents*. The end control packet is sent through *sendEndPacket*.

```c
int sendControlPacket(TLV* tlvs, int
tlvNum, u_int8_t cf);
```

Sends the file's general information. Reserves space in memory depending on *tlvNum*, creates the packet and then calls *llwrite* to send it.

```c
int sendFileContents(FILE *fp, long
size);
```

Calls llwriteto send the file *fp* of size *size* through the serial port.

```c
int sendEndPacket()
```

Sends the file's end packet to mark the end of the trasmission.

```c
int appLayer_exit();
```

Ends the application by calling *llclose* to close the serial port connection and freeing any remaining memory used.

3.2 - Data Link Layer

The Data Link Layer is the lower level component of the program, which is responsible for communicating with the serial port, established in the *link_layer.c* file.

A dynamic array implementation is also defined in *dynamic_array.c*, which includes functions for stuffing and destuffing a frame.

```
int llopen(LinkLayer
connectionParameters);
```
Opens the connection to the serial port, making use of the *connectionParameters* struct.

```
int llwrite(const unsigned char *buf,
int bufSize);
```
Writes frame in *buf* with size *bufSize* to the port given in the global variable *fd*.

```
int llread(unsigned char *packet);
```
Reads frame to *buf* from the port given in the global variable *fd*.

```
int llclose(int showStatistics);
```
Closes the connection to the serial port.

```
int sendSUFrame(int fd, LinkLayerRole
role, u_int8_t msg);
```
Sends a supervisory unnumbered frame of control field *msg* to the port given by the variable *fd*, given the struct *role* that defines the role of the function caller (sender or receiver).

```
int sendIFrame(int fd, const unsigned
char *buf, int length, int seqNum);
```
Sends an information frame from *buf* to the port given by the variable *fd*, of size *length* and sequence number *seqNum*.

```
u_int8_t readSUFrame(int fd,
LinkLayerRole role);
```
Reads a supervisory unnumbered frame from the port given by the variable *fd*, given the struct *role* that defines the role of the function caller (sender or receiver).

```
int readIFrame(int fd, unsigned char
*buf, int seqNum);
```
Reads an information frame to *buf* from the port given by the variable *fd*, of sequence number *seqNum*.

4. Main Use-Cases

The two main use cases are sending a file or receiving a file. The flow of these use cases is as follows:

Sender

- Runs the program (*main*), which starts the application (*applicationLayer*).

- *llopen* is called, and the connection to the serial port is opened. *sendFile* is called.

- *sendFile* opens the file and sends its general information (not the contents) to the receiver, and then calls *sendFileContents* with the file and its size as the parameters.

- *sendFileContents* divides the file data in packets, whose size depends on the user's choice, and sends them sequentially through *llwrite* to the receiver, until it finishes the file or or errors occur, that trigger the timeout more than the number of retries specified. These can be detected thanks to the BCCs, and along with them the header and the start/end flags are also added. The data is also stuffed to prevent misinterpretation of part of it as flags, for example.

- The file is closed, followed by the application in *appLayer_exit*, which calls *llclose*.

Reader

- Runs the program (*main*), which starts the application (*applicationLayer*).

- *llopen* is called, and the connection to the serial port is opened. *readFile* is called.

- *readFile* reads file information (except the content) and stores it in a struct called *file_info*, and then calls *writeFileContents* with the file to be received as the parameter.

- *writeFileContents* receives the information packets through *llread* which, after being destuffed and checked for errors, have their headers, flags and BCCs removed. Finally, each packet is written to the file sequentially, until every packet is received or errors occur, that trigger the timeout more than the number of retries specified.

- The file is closed, followed by the application in *appLayer_exit*, which calls *llclose*.

5. Data Link Protocol

The data link layer has 4 main functions: *llopen* (used for establishing connection at the start of the data transfer), *llwrite* used to send information frames, consequently, *llread* to read information frames and, finally, *llclose* to end the connection.

5.1 - *llopen*

```
int llopen(LinkLayer
connectionParameters);
```

Before any information is exchanged between the receiver and the transmitter, *llopen* verifies that the serial port is open and available for usage. In order to use timeouts on read and write functions, a *termios* structure is also used with values VTIME and VMIN set to *connectionParameters.timeout* * 10 and 0 respectively.

The function receives only one parameter, *connectionParameters*, that contains important information used for the function's protocol, that goes as follows:

- The transmitter emits a SET frame and waits for the receiver's response. Once an acknowledgement frame (UA frame) sent by the receiver is received, the connection has been correctly established and *llopen* closes. If no acknowledgement frame is received by the transmitter during a set timeout period (parameter defined by *connectionParameters*'s *timeout* value), it retries the protocol, sending a SET frame and waiting for an UA frame again. If this does not happen after the number of retries set by the *connectionParameters*'s *nRetransmissions* value, *llopen* ends with an error number, and the program closes.

```
else if (connectionParameters.role == LlTx) {

        while (nTries++ <
connectionParameters.nRetransmissions) {
            sendSUFrame(fd, LlTx, CTRL_SET);

            if (readSUFrame(fd, LlTx) ==
CTRL_UA) {
                printf("Received
acknowledgement frame. Continuing...\n\n");
                return 1;
            }
        }
        return -2;
    }
```

- The receiver waits for a SET frame reception and acknowledges it by sending a UA frame back.

```
if (connectionParameters.role == LlRx) {
        //wait for frame to arrive
        while (readSUFrame(fd, LlTx) !=
CTRL_SET) { }
        sendSUFrame(fd, LlTx, CTRL_UA);
    }
```

5.2 - *llwrite*

```
int llwrite(const unsigned char *buf,
int bufSize);
```

llwrite's goal is to send information to the receiver in an information frame and, because of this, is exclusively used by the transmitter (no information frame is ever sent by the receiver).

Initially, it writes to the serial port using a function called *sendIFrame*. This auxiliary function takes the serial port file descriptor (global variable in our link layer code file), the file buffer and its filesize and the current protocol's sequence number. After building an information frame with these latter 3 parameters, it writes the frame to the file descriptor. This information frame follows the same structure every time:

- a starting flag (1 octet),

- the address of the information (1 octet),

- the control field that carries the sequence number (1 octet),

- a BCC calculated through an XOR between the address and the control field (1 octet),

- every data byte contained in the information buffer being sent (buffer size octets),

- another BCC calculated through an XOR between every data octet being sent,

- and an ending flag.

To ensure the receiver always understands where the first and last flags are (delimiters of the frame), a byte stuffing mechanism is implemented, and its job is to escape any flag or escape byte contained in the frame by replacing it with two bytes: an escape byte 0x7e followed by a byte calculated through an XOR between the escaped byte and 0x20. Obviously, the delimiting flags are not stuffed.

After this, there are 3 major ways in which the program can proceed, depending on the receiver's answer:

- The transmitter receives a supervision frame with a RR control field within the previously established *ll_connectionParameters.timeout* and the next sequence number. With this, the transmitter knows the reeceiver acknowledged the information frame positively, since it deemed it correct (it contained no header errors, its sequence number is correct and there were no errors in the data portion of the frame). With this, *llwrite* closes successfully and returns the number of bytes were sent (size of the information frame).

- The transmitter receives a supervision frame with a REJ control field with an unchanged sequence number. This means the information frame sent was not acknowledged correctly and hence needs to be re-sent. The retry number is reset to 0, and the protocol restarts from scratch.

- The transmitter receives any other type of response, and re-sends the frame, awaiting a correctly acknowledged RR response.

The retry number is increased, and the protocol restarts.

This back-and-forth is wrapped in a *for* loop that repeats until the number of retries is the same as the previously established *ll_connectionParameters.nRetransmissions*. If this for loop reaches its end condition, the program returns an error number.

```
    for (int nTries = 0; nTries <
ll_connectionParameters.nRetransmissions;
nTries++) {
        //printf("Sending Iframe...\n");
        if ((bytes = sendIFrame(fd, buf,
bufSize, old_seqNum)) < 0) {
            printf("Error sending
Iframe.\n\n");
            continue;
        }

        response = readSUFrame(fd,
ll_connectionParameters.role);
        //printf("response: %d\n", response);

        if (response == CTRL_RR(next_seqNum))
{
            //printf("Iframe acknowledged
correctly. Continuing...\n\n");
            seqNum = next_seqNum;
            return bytes;
        }
        else if (response ==
CTRL_REJ(old_seqNum)) {
            printf("Iframe rejected (wrong
BCC2).\n\n");
            //seqNum = next_seqNum;
            nTries = 0;
            continue;
        }

        else if (response == 0) {
            printf("No SUframe received
(timeout).\n\n");
            continue;
        }
    }
    printf("LLWrite failure (too many
tries).\n\n");
    return -1;
```

5.3 - *llread*

```
int llread(unsigned char *packet);
```

llread's goal is to read an information frame sent by the transmitter and posteriorly fill the received *packet* parameter with the data contained inside the information frame. It is also the only place where the sequence number is changed depending on the flow of the function.

Firstly, it retrieves the data contained in an information frame using a function called *readIFrame*. This auxiliary function takes the serial port file descriptor (global variable in our link layer code file), the file buffer and its filesize and the current protocol's sequence number. The information frame is parsed inside this function following the same structure as *sendIFrame* uses to construct it, however, to ensure transparency, destuffing the frame after making sure everything was received (reading everything from the serial port from the first flag to the second one) is needed.

If the frame was received correctly with no errors detected, the return value is the number of size in octets of the data part of the information frame (without the header, delimiting flags and second BCC). If an error was detected, the error numbers range from -1 to -4, and their meanings are as follows:

- -1 signifies a read timeout

- -2 signifies an error in the frame's header

- -3 signifies an out unexpected sequence number

- -4 signifies an invalid BCC2 and, consequently, an error in the data department

After this, there are 3 major ways in which the program can continue, depending on the *readIFrame*'s return value:

- The return value was positive, meaning the frame was received correctly. In this case, *llread* sends an RR control field supervision frame using the next sequence number, changing it in the process, to alert the transmitter that the frame was both received in sequence & acknowledged positively and returns the number of size in octets of the data part of the information frame.

- The return value was -3, which means the retrieved information frame was a repeated one. *llread* sends an RR control field supervision frame using the current sequence number, leaving it unchanged, to alert the transmitter nothing of significance was done with the repeated frame, and returns the -3 error number.

- The return value was -4, which means the frame found a problem inside the data. *llread* sends an REJ control field supervision frame with the current sequence number, leaving it unchanged, that alerts the transmitter to re-send the frame with this sequence number.

```
    //if read successful, change seqNum
    int bytes;

    //printf("Reading Iframe...\n");

    bytes = readIFrame(fd, packet, seqNum);

    if (bytes < 0) {
        switch (bytes) {
            case -3:
                printf("Sending same seqNum
acknowledgement SU frame...\n\n");
                sendSUFrame(fd, LlTx,
CTRL_RR(seqNum));
                break;
            case -4:
                printf("Sending rejection SU
frame...\n\n");
                sendSUFrame(fd, LlTx,
CTRL_REJ(seqNum));
                break;
            default:
                break;
        }
    }
    else {
        seqNum = seqNum ? 0 : 1;
        //printf("Sending IFrame reception
acknowledgement SU frame...\n\n");
        sendSUFrame(fd, LlTx,
CTRL_RR(seqNum));
    }
    return bytes;
```

5.4 - *llclose*

```
int llclose(int showStatistics);
```

llclose is used after the end of the data transfer and at the end of the program's execution. Receives a parameter that defines if statistics about the data exchange are to be shown or not. Its main goal is to make sure the connection closes correctly, returning 0 if this happens, and uses the following protocol:

- The transmitter sends a DISC (disconnect) frame, and waits for the receiver to send a DISC frame back. After this condition is met, the transmitter sends an UA acknowledgement frame to the receiver. If no DISC frame is received by the transmitter during a set timeout period (parameter defined by *ll_connectionParameters*'s *timeout* value), it retries the protocol, sending a DISC frame and waiting for an DISC frame again. If this does not happen after the number of retries set by the *ll_connectionParameters*'s *nRetransmissions* value, *llclose* ends with an error number, and the program closes.

```
 else if (ll_connectionParameters.role ==
LlTx) {

        while (nTries++ <
ll_connectionParameters.nRetransmissions) {
            sendSUFrame(fd, LlTx, CTRL_DC);
            if ((msg = readSUFrame(fd, LlTx))
== CTRL_DC) {
                sendSUFrame(fd, LlTx,
CTRL_UA);
                break;
            }

        }
    }
```

- The receiver waits for a DISC frame to be sent by the transmitter and, after receiving it, sends a DISC frame back, and waits for the transmitter's acknowledgement UA frame.

```
if (ll_connectionParameters.role == LlRx) {
        while (nTries <
ll_connectionParameters.nRetransmissions) {
            if ((msg = readSUFrame(fd, LlTx))
== CTRL_DC) {
                sendSUFrame(fd, LlTx,
CTRL_DC);
                break;
            }
        }
    }
```

6. Application protocol

The application layer consists of several functions that manage the flow of the program and have various goals like connection establishment, file transfer through the serial port and connection closure when this file transfer ends.

6.1 - *applicationLayer*

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

Main starting function used at the start of the program. Every parameter this function takes will be used to construct a LinkLayer struct that *llopen* will make use of.

Returns 0 if *llopen* went through with no errors, else returns -1.

6.2 - *sendFile*

```
int sendFile(char* path);
```

This is the function the defines the transmitter's actions when exchanging data retrieved from the file during the whole program. Receives the to-be-sent file's relative path as a parameter.

Returns 0 if every step of the function returns correctly, and returns -1 if it hasn't. Its flow is as follows:

- Construct a TLV structure containing filesize data by opening the file using *fopen* on the *path* parameter and using *fseek* to calculate its size

- Construct a TLV structure containing filename data through the usage of the *basename* function on the *path* parameter it receives, as well as retrieving its filesize by means of the *'strnlen* function

- Send a starting control packet with both of these TLV's using the *sendControlPacket*

- Send the file's contents with the function *sendFileContents*

- Send ending control packet using the *sendEndPacket* function that utilizes the *startPacket* structure where the first starting packet was previously stored

- Close the file

6.3 - *sendFileContents*

```
int sendFileContents(FILE *fp, long
size);
```

This is the function the transmitter uses to send the file's contents in data packets of a user-defined maximum size until the file has ended. Receives the file descriptor pertaining to the file that is being transfered and the size of said file.

Returns 0 if the whole file was sent without any problems, otherwise returns -1.

Every data packet sent follows the same structure, and this structure is as follows:

- a control field with value 1 (used for data packets only) (1 octet)

- a sequence number field containing the sequence number mod 256 (1 octet)

- an L2 field containing the number of bytes from the file to send divided by 256 (1 octet)

- an L1 field containing the number of bytes from the file to send mod 256 (1 octet)

- data field filled with information from the file (256*L2+L1 octets)

```
    while (file_to_go > 0) {
        if (file_to_go < MAX_PACKSIZE - 4) {
            content_size = file_to_go;
        }
        // insert as much file content into
packet as possible (4 bytes will be used for
other packet fields)
        if ((read_res = fread(packet + 4, 1 ,
content_size, fp)) < 0) {
            printf("FRead error while sending
file contents. \n");
            return -1;
        }

        packet[0] = CF_DATA;
        packet[1] = (unsigned char)
((cur_seqNum++) % 256);
        packet[2] = (unsigned char) (read_res
/ 256);
        packet[3] = (unsigned char) (read_res
% 256);
        printf("%i\n", cur_seqNum);

        if ((llwrite(packet, read_res + 4)) <
0) {
            memset(packet, 0,
sizeof(packet));
            return -1;
        }
        else {
            file_to_go -= read_res;
            numDataPackets++;
            memset(packet, 0,
sizeof(packet));
        }
    }
```

6.3 - *readFile*

```
int readFile();
```

This function defines the receiver's actions during the flow of the program.

Returns 0 if every step in the function returns positively, otherwise returns -1.

The flow of the function is as follows:

- Reads the first control packet using *readControlPacket* which, while parsing the the starting packet, fills a global *file_info* structure with file information sent by the transmitter, like the filesize and the filename

- Creates a new filename from the filename received and opens a new file for writing using that filename

- Writes to this new file using the function *writeFileContents*

- Closes the file

6.4 - *writeFileContents*

```
int writeFileContents(FILE *fp);
```

This is the function the receiver uses to write the received data to a file. Receives the file descriptor of the newly-opened file as a parameter.

Returns 0 if data reception went without errors, otherwise returns -1.

6.5 - *appLayer_exit*

```
int appLayer_exit();
```

Simple function that closes the program after data exchange is done.

Exits with 0 if program ended successfully with *llclose*, otherwise exits with -1.

7. Validation

The program was validated using both the virtual serial ports and the physical ones, and ended successfully with identical files facing the following tests:

- Temporary disconnection of the serial port

- Physical interference with the serial port

- Different file format usage ("pinguim.gif", "relva.jpg", "teste.txt")

- Different maximum packet sizes (32, 128, 256, 512, 1028);

- Different baudrates;

- Different (artificial) propagation times.

8. Data Link Protocol Efficiency

In order to measure the efficiency of the protocol, we varied 3 fields: the Frame Error Ratio (FER), the Propagation Time and the Information Frame Size. The baudrate could have been changed, but through ssh it is not possible. The graphs for each test are in Annex 2.

8.1 - Baudrate

Even though it was not tested, changing the baudrate would not theoretically yield much difference in terms of efficiency, given that the efficiency is calculated by dividing the real flux of information by the baudrate, and this flux increases with the baudrate.

8.2 - Frame Error Ratio

The FER affects the efficiency in two different ways, depending on where it is increased/decreased:

- in the BCC1, increasing the FER drastically reduces the efficiency, since each error causes the program to wait for the duration of the timeout.

- in the BCC2, increasing the FER reduces the efficiency as well, albeit to a smaller, more linear extent, since each error simply prompts a retransmission.

8.3 - Propagation Time

Increasing the propagation time meant adding a usleep function before every frame read. As expected, the efficiency lowers as the propagation time increases.

8.4 - Information Frame Size

The efficiency increases with bigger information frame sizes being sent, since it requires less calls to the data link protocol functions.

9. Conclusions

This project was a successful practical application of the theory learned during classes. It was made in accordance to the specifications and, judging by the test results, is compliant with the expected outcome from the procedures. Overall, the project served as a good experience and an excellent learning opportunity.

Annex 1 - Source code

Available in the *code/* folder of the project.

Annex 2 - Graphs and other statistics

We only calculate the effects of FER in the BCC2 since the timeout is not something directly related to the performance of the data link protocol.

Default parameters

File size (bytes)	Frame Size (bytes)	Baudrate (bits/s)	Average time (s)	Flow (bits/s)	Efficiency (bits/s)
153891	512	38400	5.313	28964.992	0.754

Table 1 - Varying information frame size

Maximum frame size (bits)	Average time (s)	Efficiency
32	9.347	0.429
128	6.068	0.660
256	5.572	0.719
512	5.313	0.754
1024	5.211	0.769

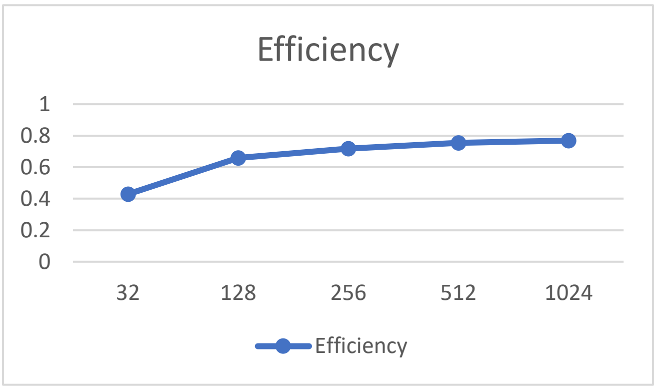


Table 2 - Varying propagation time

Distance (km)	Average time (s)	Efficiency
1	5.315	0.754
100	5.373	0.746
1000	5.901	0.679

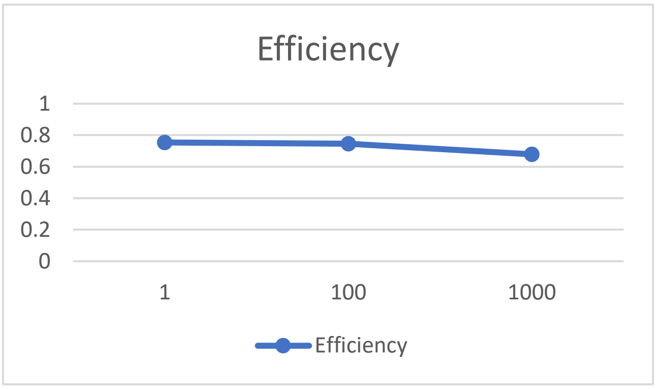
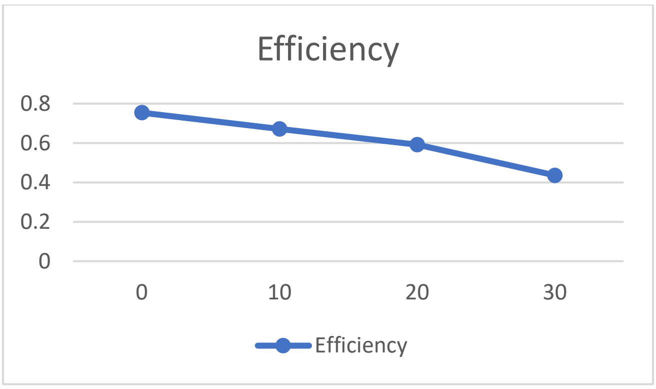


Table 3 - Varying frame error ratio

FER (%)	Average time (s)	Efficiency
0	5.313	0.754
10	5.960	0.672
20	6.774	0.592
30	9.186	0.436


Efficiency
Efficiency