Relatório do 1º Trabalho Laboratorial

Porta Série

Trabalho realizado por:

Rui Moreira **up201906355**
José Silva **up201904775**

Unidade Curricular: Redes de Computadores

GitHub Repositório com o código

1. Introdução

O principal objetivo deste trabalho é implementar um protocolo de ligação de dados que forneça um serviço de comunicação fiável entre dois sistemas ligados por cabo série e criar uma aplicação que implementa um protocolo de aplicação simples de transferência de ficheiros.

2. Arquitetura

Este trabalho está maioritariamente dividido em duas partes: o sender(emissor) e o receiver(receptor).
Ambas as partes utilizam funções da camada de ligação de dados e da camada de aplicação. No entanto, existe uma distinção entre quem está a utilizar estas camadas garantindo então a independência entre ambas as partes.

3. Estrutura de Código

O código desenvolvido para este trabalho encontra-se dividido em 5 grandes blocos. Sendo estes o sender.c que corresponde ao emissor; o receiver.c que corresponde ao recetor; dataLayer.c que corresponde à camada de dados; application.c que corresponde à camada da aplicação; serial.c que corresponde ao main() principal onde ocorrem as chamadas às funções de aplicação, dependendo de quem está a executar, o emissor ou o receptor, e consequentemente a chamada às funções de ligação de dados. *(Todo o código desenvolvido está documentado em doxygen)*

sender.c (emissor):

- **sendSetFrame(int fd) -** envio de trama de supervisão SET e receção e validação trama UA.
- **openSender(char filename[])** - abertura do file descriptor para envio de tramas, verificando a abertura correta da porta série através de *sendSetFrame(fd).*
- **senderDisc(int fd)** - envio de trama de supervisão DISC, receção e validação do trama DISC e envio de trama UA.
- **closeSender(int fd)** - fecho da porta série, utilizando a *senderDisc(fd)* para um fecho correto.
- **dataStuffing(char *buffer, int dataSize, char BCC2, char *stuffedBuffer)** - função que realiza o stuffing de dados antes do envio para a serial port.
- **sendStuffedFrame(int fd, char *buffer, int bufferSize)** - chamada a *dataStuffing(char *buffer, int dataSize, char BCC2, char *stuffedBuffer),* para realizar o stuffing de dados, antes do seu envio para o outro lado da porta série.
- **answerAlarm() && resetAlarmFlags()** - funções responsáveis pela gestão do alarme inserido para a detecção de erros, ou eventual desconexão da porta série.

receiver.c (receptor):

- **receiveSetFrame(int fd)** - receção e validação de trama de supervisão SET e envio de trama UA.
- **openReceiver(char filename[])** - abertura do file descriptor para a receção de trames, verificando a abertura correta da porta série através de *receiveSetFrame(fd)*.
- **receiverDisc(int fd)** - receção e validação de trame de supervisão DISC, envio do trama DISC, e consequentemente se todos os passos anteriores estiverem corretos receção e validação de trama UA.
- **closeReceiver(int fd)** - fecho da porta série, recorrendo a receiverDisc(fd) para validação de fecho em ambos os lados da porta série

- **dataDeStuffing(char *stuffedBuffer, int stuffedBufferSize, char *buffer, char *BCC2)** - *destuffing* da data recebida pela porta série, e consequente verificação da integridade dos dados.
- **receiveStuffedBuffer(int fd, char *buffer)** - receção de data stuffed da porta série, e após *dataDeStuffing(char *stuffedBuffer, int stuffedBufferSize, char *buffer, char *BCC2),* caso a integridade dos dados não seja verificada, envio de trama com deteção do erro.
- **receiveStuffedBufferSM(MACHINE_STATE *state, int fd, char *stuffedBuffer, char *buffer)** - state machine para verificar a integridade do *stuffed buffer*.

dataLayer.c (Camada de ligação de dados):

- **llopen(char *filename, int id)** - Abertura da porta série, dependendo do lado que a está a abrir *emissor ou recetor.*
- **llclose(int fd, int id)** - Fecho da porta série, dependendo do lado que quer fechar *emissor ou recetor.*
- **llwrite(int fd, char *buffer, int size, int id)** - Envio de dados pela porta série.
- **llread(int fd, char *buffer, int id)** - Receção de dados pela porta série.

application.c (Camada da aplicação):

- **sendControlPacket(int fd, u_int8_t cField, long fSize, char fName[ ])** - Cria e envia os pacotes de controlo.
- **readControlPacket(int fd, u_int8_t cField, u_int8_t buf[ ], char** fName, long fSize)** - Lê um pacote de controlo e aloca o nome e o tamanho do ficheiro.
- **dataPacketBuilder(u_int8_t* dPacket,int dPacketSize, u_int8_t* fData, int fDataSize, u_int8_t sequence)** - Cria um pacote de dados a partir de um buffer de dados.
- **sendData(int fd, FILE* fd1, long fSize)** - Envia os pacotes de dados.
- **sendFile(int fd, char fPath[ ])** - Envia os pacotes de controlo e os pacotes de dados.
- **readFile(int fd)** - Lê todos os pacotes e cria o ficheiro recebido.

common.c (Código utilizado em vários módulos):

- **checkSupervisionFrame(MACHINE_STATE *state, int fd, char A_BYTE, char C_BYTE, char *reject)** - state machine para verificar a integridade de tramas de supervisão do tipo *UA, DISC e SET*.
- **getByteFromFD(int fd, char *byte_to_be_read)** - função auxiliar para ajudar na leitura byte a byte do file descriptor da camada de ligação de dados.
- **sendSupervisionFrame(int fd, char A_BYTE, char C_BYTE)** - envio de tramas de supervisão do tipo *UA,SET e DISC.*
- **createBCC2(char *buffer, int bufferSize, char *bcc2)** - criação do BCC para a trama de informação.
- **insertError(char *data, int size, int probability)** - inserção de um erro de acordo com uma probabilidade para testar a robustez do código desen

serial.c (Interface e main()):

- **parseArgs(int argc, char **argv, char *id, char *file, char *serialport)** - dar parse aos inputs fornecidos no terminal para a execução do programa, obedecer às instruções dadas pelo utilizador.
- **printHelpMessage()** - imprimir no ecrã do terminal, as regras de utilização do comando seriaport.
- **validateArgs(int argc, char **argv, char *id, char *file, char *serialport)** - validação do input fornecido pelo utilizador.
- **execution(int argc, char **argv)** - Execução do programa de acordo com o input válido dado pelo utilizador, existindo distinção entre *emissor e recetor.*

macros.h (macros pertinentes utilizadas em vários módulos):

- **ALARM_INTERVAL** - tempo do alarme, para gestão de erros ou desconexão.
- **MAX_NO_ANSWER** - número máximo de timeouts antes de desconexão.

- **BIT(n) 1 << n** - Bit Mask, para ativar o bit na posição n.
- **C_FRAME_I(n) (BIT(6*(n)) & 0x40) -** C field para trama de informação
- **C_RR(n) (BIT(7*(n)) | 0x05)** - C field para RR byte
- **C_REJ(n) (BIT(7*(n)) | 0x01)** - C field para REJ byte
- **ESCAPE** - escape byte
- **FLAG_ESCAPE_XOR** - FLAG xor 0x20
- **ESCAPE_XOR** - escape xor 0x20
- **BCC(a,c)** - bcc field gerador
- **FLAG** - flag byte
- **A_SR 0x03** - A field Emissor para recetor
- **A_RS 0x01** - A field Recetor para emissor
- **C_UA 0x07** - C field para tramas UA
- **C_SET 0x03** - C field para tramas SET
- **C_DISC 0x0B** - C field para tramas DISC
- **MAX_DATA_SIZE** - tamanho máximo dos dados a enviar
- **STUFF_DATA_MAX** - tamanho máximo dos dados a enviar depois de stuffing
- **MACHINE_STATE** - enum que contém todos os estados possíveis da máquina de estados

4. Casos de uso principais

Interface

A interface desenvolvida permite ao utilizador, escolher se quer ser o emissor ou o recetor, escolher qual o device driver correspondente à porta série no seu dispositivo, escolher o ficheiro que quer enviar, e até ativar o modo de debug.

Utilizando a consola, correndo ./serialport -h, terá acesso a uma mensagem de ajuda com a explicação da utilização do programa.

```
rui@rui-Legion-Y540-15IRH:~/Desktop/FEUP-Pratical-Projects-2021/RCOM/RCOM-Pratical/proj1$ ./serialport -h
Usage: serialport [ID]... [OPTION] ... [FILE]...
Establish serial port connection to send [FILE] or receive [FILE]

If file path is not given, pinguim.gif will be sent/received by default, if available on execution root folder

        -d                      enable debug mode, output written to files stdout.txt stderr.txt, respectively
        -i                      serial port side identifier, sender || receiver
        -h                      serial port help message
        -f                      file to sent/received through the serial port, being ./pinguim.gif the default value if no arguments are provided

Examples:
        serialport -i sender -d -f ./file.txt                   Serial port sender side, debug mode active sending file.txt
        serialport -d -i receiver                               Serial port receiver side, debug mode active receiving pinguim.gif
```

Transmissão de dados

Ocorre através da porta série, entre dois computadores, gerando-se a seguinte sequência de eventos:

a. O emissor, escolhe o ficheiro a ser enviado;
b. A ligação entre os computador do emissor e do recetor é configurada;
c. Após a configuração e validação da ligação é estabelecida;
d. Os dados são enviados pelo emissor trama a trama, de acordo com o tamanho definido;
e. Consequentemente, o receptor recebe e valida os dados trama a trama;
f. Com o envio dos tramas, o receptor vai guardar os tramas já recebidos num ficheiro com o mesmo nome do ficheiro enviado pelo emissor;
g. Após o envio total do ficheiro sem nenhum tipo de erro, a ligação é terminada.

5. Protocolo de Ligação Lógica

De acordo com o guião do trabalho, foram implementados na camada lógica as funções **llopen, llclose, llwrite** e **llread.** Servem de *interface* para o protocolo de aplicação poder usar as funcionalidades da data layer. Respetivamente, as funções referidas anteriormente, estabelecem a

ligação entre o emissor e o recetor; interrupção de modo corretor da ligação entre o emissor e o recetor; envio de uma trama de informação pelo o emissor, até esta ter sucesso, ou abortar por exceder o número máximo de tentativas; e a receção de uma trama de informação, até esta ter sucesso. As leituras e consequente validação de tramas quer de supervisão quer de informação, são feitas por uma ***state machine,*** que recebe informação byte a byte e vai interpretando e desempenhando mudanças de estado, de modo a que apenas uma mensagem válida possa chegar ao estado final e consequentemente ser aceita como fidedigna.

llopen

Esta função cria a ligação à porta série, de acordo com um ***id,*** para fazer a distinção entre emissor e recetor, consequentemente fazendo a chamada a ***openReceiver*** ou a ***openSender,*** dependendo do id, que tratam do resto das funcionalidades do ***llopen***.

```
/**
 * @brief Open serial port connection through a file descriptor, in a specific
 * side of the serial port, given by the id
 *
 * @param  filename              file path to be transmmited
 * @param  id                    Receiver -> 1 | Sender -> 0
 * @return          file descriptor upon sucess, ERROR otherwise
 */
int llopen(char *filename, int id);
```

openReceiver

Espera pela receção de trama *SET*, e consequentemente, após validação do trama envio de trama *UA*. Para desta forma validar e estabelecer a ligação entre emissor e recetor.

```
/**
 * @brief Opens sender for sending data
 *
 * @param  filename             Serial port filename
 * @return Serial port fd upon success, ERROR otherwise
 */
int openSender(char filename[]);
```

openSender

Envio de trama de *SET*, e consequentemente, espera por um trama *UA,* ocorrendo timeout e/ou saída do programa se o número máximo de tentativas de retransmissões for ultrapassado.

```
/**
 * @brief Opens the receptor for reading and sending data
 *
 * @param  filename          serial port file
 * @return       File descriptor of the serial port upon sucess, ERROR otherwise
 */
int openReceiver(char filename[]);
```

As tramas de supervisão são criadas pela função ***sendSupervisionFrame,*** e a receção é feita através da função ***receiveSetFrame.*** A receção dos tramas de supervisão é feita byte a byte, e a sua validação é feita na função ***checkSupervisionFrame.***

llclose

Tal como na função ***llopen***, estão função recebe um ***id***, que faz a distinção entre emissor e o recetor. Tem com principal função finalizar a ligação entre os dois.

```
/**
 * @brief Closes the serial port connection, in a specific
 * side of the serial port, given by the id
 *
 * @param  fd              File descriptor of filename(llopen) file
 * @param  id              Receiver -> 1 | Sender -> 0
 * @return          SUCCESS upon success, ERROR otherwise
 */
int llclose(int fd, int id);
```

closeReceiver

O recetor recebe um *DISC,* valida-o e consequentemente envia um *DISC,* para o emissor, e caso este seja validado pelo emissor, recebe um *UA*.

```
/**
 * @brief Closes the Receiver side of the serial port (slide 14)
 *
 * @param  fd                 serial port file descriptor
 * @return    SUCCESS upon success, ERROR otherwise
 */
int closeReceiver(int fd);
```

closeSender

O emissor envia um *DISC,* consequentemente recebe um *DISC,* do recetor e após validação, envia um *UA.*

```
/**
 * @brief Closes the Sender side of the serial port (slide 14)
 *
 * @param  fd               serial port file descriptor
 * @return    SUCCESS upon success, Error otherwise
 */
int closeSender(int fd);
```

Caso as funções anteriormente destacadas não retornem *ERROR,* a ligação é fechada. Foi também implementado para estas funções a possibilidade de timeouts e/ou retransmissão de tramas. Para envio do trama de supervisão tal como em **llopen,** é usada a função ***sendSupervisionFrame, senderDisc, receiverDisc*** que são funções auxiliares que ajuda no envio e verificação de tramas de supervisão *DISC e UA*.

llwrite

```
/**
 * @brief Sends a packet with the data , the data can't extend the specified max length
 *
 * @param  fd                  File descriptor of the RS-232 port
 * @param  buffer              Data to be written to serial port
 * @param  size                Buffer size
 * @param  id                  Receiver -> 1 | Sender -> 0
 * @return        Data size upon sucess, ERROR otherwise
 */
int llwrite(int fd, char *buffer, int size, int id);
```

Esta função é apenas chamada pelo lado do emissor, mas a verificação se é chamada no lado do recetor é feita, e caso se verifique o programa termina com um erro. Essencialmente, consiste no envio de tramas de informação para o recetor, e é composta por várias etapas:

a. Receção de trama de informação, utilizando os dados fornecidos pelo lado da aplicação;
b. *Stuffing* dos trama de informação recebido (***dataStuffing*** function);
c. Envio de trama, com o número correto de sequência;
d. Leitura da resposta, com possibilidade de timeout;
e. Saída da função, caso seja recebido um RR, ou reenvio da trama, se for recebido um REJ, as vezes que forem necessárias até receber uma confirmação de sucesso no reenvio.

```
resetAlarmFlags(); /*Upon sending the SET FRAME reset the alarm flags, upon checking for receiver timeout*/

MACHINE_STATE setState = START_;

char setFrame[5] = {FLAG, A_SR, C_SET, BCC(A_SR,C_SET),FLAG};

while(setState != STOP_){
        if( conta == MAX_NO_ANSWER ){
                        fprintf(stderr,"Communication between Receiver && Sender failed SET\n");
                        return ERROR;
        }

        if( flag ){
                flag = 0; /* Disable message send flags */
                if( write(fd, setFrame, 5) == -1 ){
                        fprintf(stderr,"Error writing to Serial Port SET trame\n");
                        return ERROR;
                }
                setState = START_;
                alarm(ALARM_INTERVAL);
        }

        if( checkSupervisionFrame(&setState, fd, A_SR, C_UA, NULL) == ERROR) return ERROR;  /* Getting information byte by byte */
}

if(DEBUG) fprintf(stdout,"Sucessfully got UA response from receiver\n");

alarm(0); /* Disconnect alarm */

return SUCCESS;
```

O timeout das tramas *I,SET,DISC e UA* é feito da seguinte forma. **flag e conta** são flags que são modificadas pelo alarm handler, por nós definido, sempre que ocorre um timeout, que indicam se o programa de acabar ou tentar reenviar o trama, respetivamente. (exemplo do tratamento de timeout/reenvio dos tramas na imagem seguinte)

llread

```
/**
 * @brief Receives packet with the data, sent throw the serial port
 *
 * @param  fd                  File descriptor of the RS-232 serial port
 * @param  buffer              Buffer with data read from the serial port
 * @param  id                  Receiver -> 1 | Sender -> 0
 * @return        Data size upon sucess, ERROR otherwise
 */
int llread(int fd, char *buffer, int id);
```

Esta função só pode apenas ser chamada pelo recetor, mas a verificação se é chamada do lado do emissor é feita, e caso acontece o programa termina com um erro. Essencialmente, consiste em receber tramas de informação por parte do emissor, e é composta pelas seguintes etapas:

a. Leitura da trama de informação enviada pelo emissor (**receiveStuffedFrameSM);**
b. Verificação da integridade da trama;
c. *Byte destuffing* da trama de informação recebida (**dataDeStuffing**);
d. Verificação e comparação do BCC2 da trama recebida com o gerado em *runtime;*

e. Tendo em conta os passos anteriores, e a respostas obtidas, fornecer uma resposta ao emissor;
f. Construção e consequente envio da trama de supervisão correta ao emissor(RR ou REJ, devidamente acompanhados do número de sequência correto correspondente ao trama correto)
g. Caso todos os passos tenham sido executados normalmente, saída da função. Caso algum problema for detetado, há tentativa de nova leitura do trama.

6. Protocolos de aplicação

As duas funções principais da camada de aplicação são as funções **readFile** e **sendFile** usadas pelo o receptor e pelo emissor na transferência de dados, respetivamente. Estas duas funções são encarregues por executar todas as funções necessárias em cada máquina.

readFile

Esta função tem como principais funcionalidades:

```
/**
 * @brief Reads all packets and creates the file read
 *
 * @param  fd                       File descriptor
 * @return    file Size upon success, otherwise -1
 */
int readFile(int fd);
```

- Recepção do pacote de controlo de START.
- Abertura de um ficheiro com um nome igual ao que foi passado no pacote START.
- Recepção, pacote a pacote, utilizando o llread, de cada fragmento do ficheiro recebido.
- Verificação do número de sequência de cada pacote recebido.
- Escrita de cada fragmento do ficheiro recebido no ficheiro criado.
- Recepção do pacote de controlo END.

A função readControlPacket foi utilizada para ler o pacote de controlo START.

sendFile

Esta função tem como principais funcionalidades:

```
/**
 * @brief Sends the controlPackets and the data packet
 *
 * @param  fd                       File descriptor
 * @param  fPath                    File Path
 * @return          file Size upon success, otherwise -1
 */
int sendFile(int fd, char fPath[]);
```

- Abertura do ficheiro a ser enviado.
- Envio do pacote de controlo START.
- Envio dos pacotes de dados.
- Envio do pacote de controlo END.

A função sendControlPacket foi utilizada para para enviar os pacotes Start e End.
A função sendData foi utilizada para fazer dividir o ficheiro a enviar, criar os pacotes de dados e enviá-los.

7. Validação

Testes efetuados:

- Interrupção da ligação física da porta série

- Geração de ruído na ligação da porta série
- Envio de ficheiros com diversos tamanhos (mínimo: 11KB máximo: 500 mB)
- Envio de ficheiros com diferentes valores para o tamanho das tramas de informação
- Envio de ficheiros com variação do valor de baudrate
- Envio de ficheiros com geração de erros aleatórios
- Envio de ficheiros com a simulação de diferentes tempos de propagação das tramas de informação

Todos os testes efetuados, foram concluídos com sucesso, tendo verificado a robustez do protocolo implementado

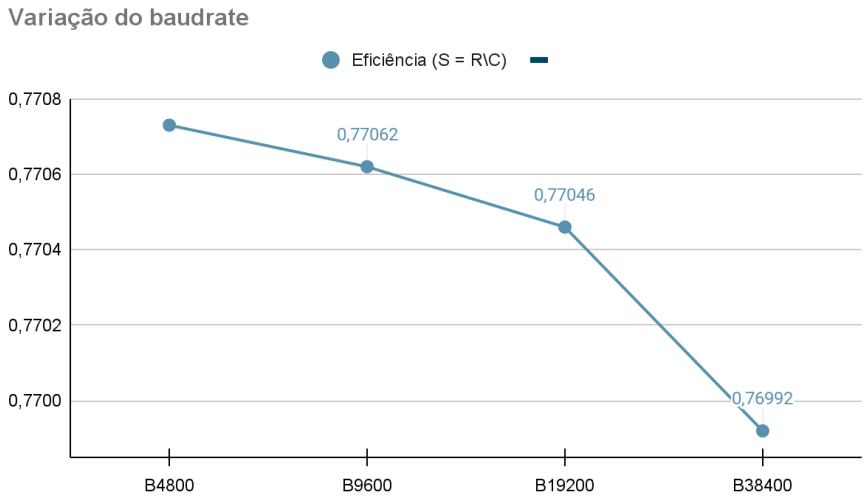
8. Eficiência do protocolo de ligação de dados

De modo a averiguar a eficiência e a robustez do protocolo desenvolvido, foram testes nas seguintes condições:

Foram executados testes com um ficheiro com cerca de 152KB, de forma a garantir que o número de tramas enviados fosse suficientemente grande para tornar o efeito das variações de variáveis testadas o mais homogêneo possível.

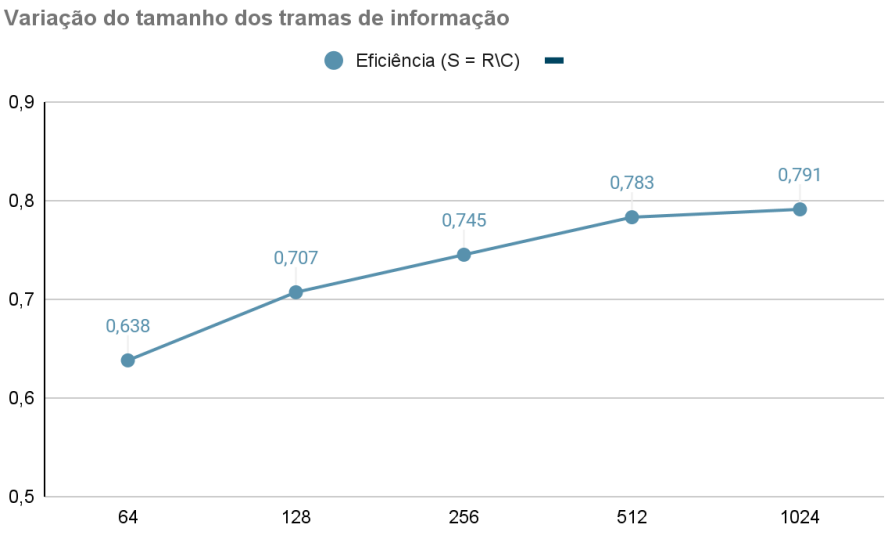
Variação do baudrate

Através da análise do gráfico pode se concluir que, com o aumento do baudrate, diminui a eficiência do protocolo.



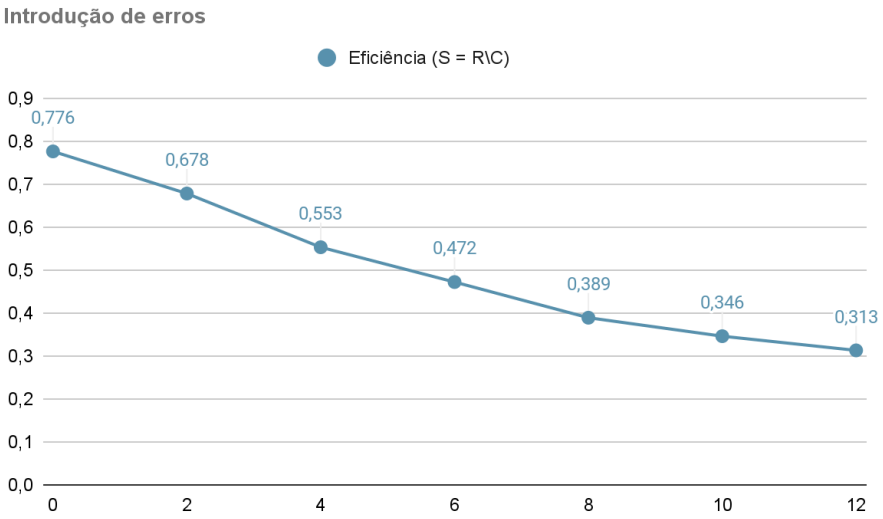
Variação do tamanho das tramas de informação

Corresponde à variação de S, de acordo com a variação do tamanho do pacote das tramas de informação, que são enviadas e recebidas. Passando à análise do gráfico, o programa será mais eficiente tanto quanto maior for o tamanho de cada pacote de dados enviado. Sendo isto lógico, pois reduz assim, os envios necessários, às verificações de integridade necessárias e a quantidade de timeouts que ocorrem durante a correta execução do programa.



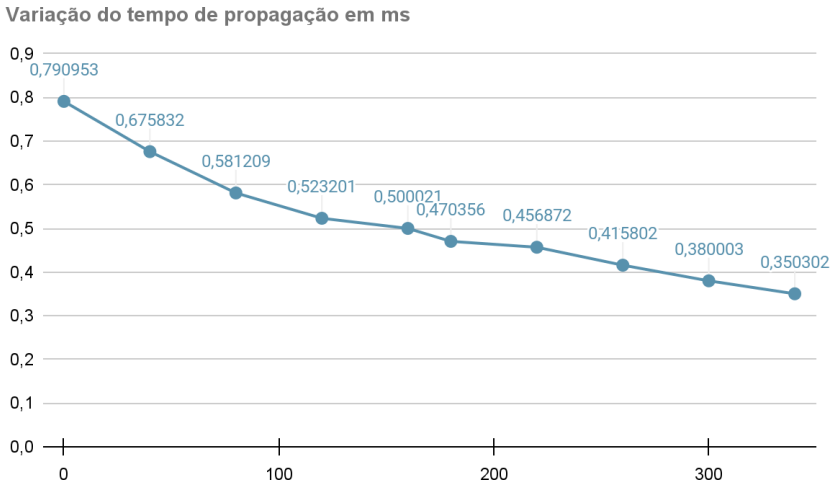
Introdução de erros dissimulados

Podemos concluir através da análise do gráfico abaixo, que a introdução de erros nos campos *BCC1* e *BCC2* tem um impacto enorme na eficiência do protocolo. Deve-se primariamente, ao simples facto de que gerando um erro no campo *BCC1,* normalmente será gerado um timeout, que tem como objetivo detetar a ausência de resposta por parte do recetor, por um número estipulado de segundos, o que afeta significativamente o tempo de execução. Pelo contrário, erros no *BCC2* não são tão impactantes negativamente, pois geram o reenvio da trama, que é mais eficiente que um timeout, pois este último é imediato.



Variação do tempo de propagação

Com o aumento do tempo de propagação de cada trama de informação, tal como seria de se esperar, o programa torna-se mais ineficiente. Devendo-se ao facto de o lado da aplicação passar mais tempo em “espera”, sem enviar tramas, uma vez que o envio/receção de tramas é mais demorado. Para obter os valores para construção deste gráfico, foi usada a função *nanosleep(),* para *simular* um aumento no tempo de propagação da trama, estando os valores entre 0 ms, até 340 ms.



Do ponto de vista teórica e tendo em conta o protocolo *Stop & Wait,* o emissor após o envio de uma trama de informação, espera da confirmação por parte do recetor *ACK* (Acknowledgment). Do lado do recetor, aquando receção de uma trama, sem erros, confirma com o envio de um *ACK,* caso contrario um *NACK (negative acknowledgment)*. Repetindo se o processo até o envio total do ficheiro. Caso uma trama de informação ou as resposta de acknowledgment não sejam recebidas, deve ser implementado para permitir o bom funcionamento do programa um mecanismo de timeouts, levando à retransmissão da trama de informação.

Para o recetor conseguir identificar a repetição da trama, as tramas de informação são numeradas, com um 0 ou um 1, sendos estes valores alternadamente utilizados. Um *ACK* com valor 1 tem de necessariamente corresponder a uma trama com valor 1.

Nós decidimos implementar o mecanismo de *Stop & Wait* para o controlo de erros, durante a execução. As tramas de informação são identificadas com 1 ou 0, e a resposta do recetor, pode conter o byte RR (*ACK) ou* REJ(*NACK).*

Trama 0 -> Erro: REJ(r = 0), Sem Erro: RR(r = 1)
Trame 1 -> Erro: REJ(r = 1), Sem Erro: RR(r = 0)

9. Conclusões

Este trabalho permitiu-nos compreender melhor o protocolo de ligação de dados incidindo sobre o processo de encapsulamento e estrutura de tramas como também a receção e envio de informação. É de destacar também a importância entre a independência entre camadas que é respeitada por cada camada do nosso trabalho. A camada de ligação de dados não recorre a nenhum recurso da camada de aplicação e esta última não conhece qualquer implementação da primeira camada.

O trabalho foi concluído com sucesso uma vez que foram alcançados todos os objetivos inicialmente propostos.

A formatação e o aspecto estético deste relatório não é o melhor devido às restrições de tamanho.

Anexo - Código Fonte

sender.c

```c
#include "sender.h"


MACHINE_STATE senderState;

int conta = 0, flag = 1, s = 0;

struct termios oldtio;


void answerAlarm(){
    flag = 1; conta++;
    fprintf(stderr, "Timeout\n");
}


void resetAlarmFlags(){
    flag = 1; conta = 0;
}


int openSender(char filename[]){
    int fd;
    struct termios newtio;


    (void)signal(SIGALRM, answerAlarm); /*Setup alarm for checking interval of non-response by the receiver*/

    /*
    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(filename, O_RDWR | O_NOCTTY);
  if (fd < 0){
    fprintf(stderr,"%s\n",filename);
    return ERROR;
  }

  if (tcgetattr(fd, &oldtio) == -1){ /* save current port settings */
    fprintf(stderr,"tcgetattr\n");
    return ERROR;
  }

  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;

  newtio.c_cc[VTIME] = 1; /* unblocks after 0.1s and after 1 char is read */
  newtio.c_cc[VMIN] = 0;


  tcflush(fd, TCIOFLUSH);

  if (tcsetattr(fd, TCSANOW, &newtio) == -1)
  {
    fprintf(stderr,"tcsetattr\n");
    return ERROR;
  }

    if( sendSetFrame(fd) < 0 ) return ERROR;

  return fd;
}


int sendSetFrame(int fd){
    resetAlarmFlags(); /*Upon sending the SET FRAME reset the alarm flags, upon checking for receiver timeout*/

    MACHINE_STATE setState = START_;

    char setFrame[5] = {FLAG, A_SR, C_SET, BCC(A_SR,C_SET),FLAG};

    while(setState != STOP_){
        if( conta == MAX_NO_ANSWER ){
                fprintf(stderr,"Communication between Receiver && Sender failed SET\n");
                return ERROR;
        }

        if( flag ){
            flag = 0; /* Disable message send flags */
            if( write(fd, setFrame, 5) == -1 ){
                fprintf(stderr,"Error writing to Serial Port SET trame\n");
                return ERROR;
            }
            setState = START_;
            alarm(ALARM_INTERVAL);
        }

        if( checkSupervisionFrame(&setState, fd, A_SR, C_UA, NULL) == ERROR) return ERROR;  /* Getting information byte by byte */
    }

    if(DEBUG) fprintf(stdout,"Sucessfully got UA response from receiver\n");

    alarm(0); /* Disconnect alarm */

    return SUCCESS;
}
```

```c
int senderDisc(int fd){
    resetAlarmFlags();  /* Reset alarm flags */

    char frame[5] = {FLAG, A_SR, C_DISC, BCC(A_SR,C_DISC), FLAG};

    MACHINE_STATE senderState = START_;

    while( senderState != STOP_ ){
        if( conta == MAX_NO_ANSWER ){
            fprintf(stdout,"Communication between Receiver && Sender failed DISC\n");
            return ERROR;
        }

        if( flag ){
            flag = 0;
            if( write(fd, frame, 5) == ERROR ){
                fprintf(stderr,"Error writing to Serial Port DISC frame\n");
                return ERROR;
            }
            senderState = START_;
            alarm(ALARM_INTERVAL);
        }

        if( checkSupervisionFrame(&senderState, fd, A_SR, C_DISC, NULL) == ERROR) return ERROR;
    }

    alarm(0); /* Disable alarm, frame read correctly */

    if(sendSupervisionFrame(fd, A_SR, C_UA) == ERROR) return ERROR;

    return SUCCESS;

}


int closeSender(int fd){
    if( senderDisc(fd) == ERROR ) return ERROR;

    sleep(1);

    if (tcsetattr(fd, TCSANOW, &oldtio) == -1)
  {
    fprintf(stderr,"tcsetattr\n");
    return ERROR;
  }
  close(fd);
  return SUCCESS;
}


int dataStuffing(char* buffer, int dataSize, char BCC2, char* stuffedBuffer){
    int temp = 0; /*Temporary variable used to insert the stuffed data inside the buffer and then used to return has the stuffedBuffer size*/

    /*Data Stuffing before BCC2 */
    for(int i = 0; i < dataSize; i++){
        if( buffer[i] == FLAG ){
            stuffedBuffer[temp++] = ESCAPE;
            stuffedBuffer[temp++] = FLAG_ESCAPE_XOR;
        }
        else if( buffer[i] == ESCAPE ){
            stuffedBuffer[temp++] = ESCAPE;
            stuffedBuffer[temp++] = ESCAPE_XOR;
        }
        else stuffedBuffer[temp++] = buffer[i];
    }

    /*BCC2 Stuffing*/
    if( BCC2 == FLAG ){
        stuffedBuffer[temp++] = ESCAPE;
        stuffedBuffer[temp++] = FLAG_ESCAPE_XOR;
    }
    else if( BCC2 == ESCAPE ){
        stuffedBuffer[temp++] = ESCAPE;
        stuffedBuffer[temp++] = ESCAPE_XOR;
    }
    else stuffedBuffer[temp++] = BCC2;


    return temp;
}
```

```c
int sendStuffedFrame(int fd, char* buffer, int bufferSize){
    resetAlarmFlags(); /* Resetting alarm variables to send stuffed byte */

    if( bufferSize > MAX_DATA_SIZE ){
        fprintf(stderr,"Buffer Size exceeded the var(MAX_DATA_SIZE) value\n");
        return ERROR;
    }

    char I_C_BYTE = C_FRAME_I(s);
    char BCC2;

    if( createBCC2(buffer, bufferSize, &BCC2) == ERROR ) {
        fprintf(stderr,"Error generating BCC2, data field problem\n");
        return ERROR;
    }

    char frameH[4] = {FLAG, A_SR, I_C_BYTE, BCC(A_SR,I_C_BYTE)};
    char frameT = FLAG;
    char stuffedBuffer[STUFF_DATA_MAX];

    int stuffedBufferSize = dataStuffing(buffer, bufferSize, BCC2, stuffedBuffer);
    //insertError(stuffedBuffer, 0, 2);
    //insertError(frameH,3,2);

    MACHINE_STATE stuffedBufferState = START_;

    while( stuffedBufferState != STOP_ ){
        if( conta == MAX_NO_ANSWER ){
            fprintf(stdout,"Communication between Receiver && Sender failed Stuffed frame\n");
            stuffedBufferState = STOP_;
            return ERROR;
        }

        if( flag ){
            flag = 0;

            if( write(fd , frameH, 4) == ERROR ){
                fprintf(stderr,"Error writing to Serial Port [Frame Header]\n");
                return ERROR;
            }

            if( write(fd, stuffedBuffer, stuffedBufferSize) == ERROR ){
                fprintf(stderr,"Error writing to Serial Port [Stuffed Data]\n");
                return ERROR;
            }

            if( write(fd, &frameT, 1) == ERROR ){
                fprintf(stderr,"Error writing to Serial Port [Frame Tail]\n");
                return ERROR;
            }

            stuffedBufferState = START_;
            alarm(ALARM_INTERVAL);
        }

        char rejB = C_REJ(s);

        int supervisionRes = checkSupervisionFrame(&stuffedBufferState, fd, A_SR, C_RR(1-s), &rejB );

        if(  supervisionRes == ERROR){
            fprintf(stderr,"Error receiving correct info from receiver\n");
            return ERROR;
        }
        else if ( supervisionRes > 0 ){
            flag = 1; conta++; /* Stuffed message wasn't acknowledged correctly resend frame */
        }

    }

    s = 1-s;

    alarm(0); /* Deactivate alarm message was sent and Acknowledgement was verified and validated correctly */

    return bufferSize;
}
```

sender.h

```c
#ifndef SENDER_H
#define SENDER_H

#include <signal.h>
#include "macros.h"
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#include "common.h"
#include <string.h>
#include <stdlib.h>
#include <unistd.h>


#define BAUDRATE B38400
#define _POSIX_SOURCE 1 /* POSIX compliant source */
#define FALSE 0
#define TRUE 1

/**
 * @brief Answer the alarm to if necessary resend the info from the sender,
 * upon not receiving any response from the receiver
 */
void answerAlarm();

/**
 * @brief Resetting alarm flags such as conta and flag to the original values,
 * if a message is correctly received by the sender
 */
void resetAlarmFlags();

/**
 * @brief Opens sender for sending data
 *
 * @param  filename            Serial port filename
 * @return Serial port fd upon success, ERROR otherwise
 */
int openSender(char filename[]);

/**
 * @brief Sending SET trame message, to setup the connection between Sender && Receiver, (slide 7 && 14)
 * expected to receive Unknow Acknowledgement from the Receiver
 *
 * @param  fd              serial port file descriptor
 * @return  SUCCESS upon success, ERROR otherwise
 */
int sendSetFrame(int fd);

/**
 * @brief Closes the Sender side of the serial port (slide 14)
 *
 * @param  fd              serial port file descriptor
 * @return    SUCCESS upon success, Error otherwise
 */
int closeSender(int fd);

/**
 * @brief Issue a DISC Frame, followed by a check of a DISC Frame sent by receiver side,
 *if the DISC frame of receiver side is valid, send a UA Frame to receiver (slide 14)
 *
 * @param  fd              serial port file descriptor
 * @return    SUCCESS upon success, ERROR otherwise
 */
int senderDisc(int fd);

/**
 * @brief Data stuffing mechanism, the stuffedBuffer is the result of the stuffint of buffer (slide 13)
 *
 * @param  buffer                   buffer before stuffing
 * @param  dataSize                 buffer size
 * @param  BCC2                     bcc2
 * @param  stuffedBuffer            buffer after being stuffed
 * @return        stuffedBuffer size upon success, ERROR otherwise
 */
int dataStuffing(char* buffer, int dataSize, char BCC2, char* stuffedBuffer);

/**
 * @brief Sends a Frame with the data stuffed to the other side of the serial Port (slide 14 && 7)
 *
 * @param  fd                     file descriptor of the serial port
 * @param  buffer                 data to be stuffed and then sent
 * @param  bufferSize             buffer size
 * @return           buffer size upon success, ERROR otherwise
 */
int sendStuffedFrame(int fd, char* buffer, int bufferSize);

#endif
```

receiver.c

```c
#include "receiver.h"

int r = 1;

struct termios oldtiosreceiver;

int openReceiver(char filename[]){
  int fd;

  struct termios newtio;

  fd = open(filename, O_RDWR | O_NOCTTY);
  if (fd < 0){
    fprintf(stderr,"%s\n",filename);
    return ERROR;
  }

  if (tcgetattr(fd, &oldtiosreceiver) == -1){ /* save current port settings */
    fprintf(stderr,"tcgetattr\n");
    return ERROR;
  }

  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;

  newtio.c_cc[VTIME] = 0; /* inter-character timer unused */
  newtio.c_cc[VMIN] = 1;  /* reading 1 byte at time */

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

  tcflush(fd, TCIOFLUSH);

  if (tcsetattr(fd, TCSANOW, &newtio) == -1){
    fprintf(stderr,"tcsetattr\n");
    return ERROR;
  }

  if( receiveSetFrame(fd) < 0 ) return ERROR;

  return fd;
}


int receiveSetFrame( int fd ){
  MACHINE_STATE receiverState = START_;

  while( receiverState != STOP_){
    if( checkSupervisionFrame(&receiverState, fd, A_SR, C_SET, NULL) < 0 ) return ERROR;  /*Check if set frame was read correctly*/

  }
  if( DEBUG ) fprintf(stdout,"Frame Sucessfully read from receiver\n");

  /* Send UA response to Sender */

  if( sendSupervisionFrame(fd, A_SR, C_UA) < 0 ) return ERROR;

  return SUCCESS;
}

int closeReceiver(int fd){
  if( receiverDisc(fd) == ERROR ) return ERROR;

  sleep(1); // Avoid changing config before sending data (transmission error)

  tcsetattr(fd,TCSANOW,&oldtiosreceiver);
  close(fd);
  return SUCCESS;
}


int receiverDisc(int fd){
  MACHINE_STATE receiverState = START_;

  if(DEBUG) fprintf(stdout,"Receiving DISC\n");
  while( receiverState != STOP_ ) {
    if( checkSupervisionFrame(&receiverState, fd, A_SR, C_DISC, NULL) == ERROR) return ERROR;
  }

  if( sendSupervisionFrame(fd, A_SR, C_DISC) == ERROR) return ERROR;

  if(DEBUG) fprintf(stdout,"Receiving UA\n");
  receiverState = START_;
  while( receiverState != STOP_ ) {
    if( checkSupervisionFrame(&receiverState, fd, A_SR, C_UA, NULL) == ERROR) return ERROR;
  }

  return SUCCESS;
}
```

```c
int receivedStuffedDataSM(MACHINE_STATE *state, int fd, char *stuffedBuffer, char *buffer){

  u_int8_t receivedAddress, receivedControl, calculatedBCC,
           ctrl = C_FRAME_I(1-r), repeatedCtrl = C_FRAME_I(r), calculatedBCC2, bcc2;

  int currentDataIdx, isRepeated, dataSize;
  u_int8_t stuffed_data[STUFF_DATA_MAX];

  while (*state != STOP_) {
    u_int8_t byte;

    if( getBytefromFd(fd, &byte) == ERROR );

    switch (*state) {
      case START_:
        if (DEBUG == 1) fprintf(stdout,"Entered in START_ | ");
        if (byte == FLAG) *state = FLAG_RCV;
        break;

      case FLAG_RCV:
        if (DEBUG == 1) fprintf(stdout,"Entered in FLAG_RCV | ");
        if (byte == FLAG) continue;
        else if (byte == A_SR) {
          isRepeated = 0;
          receivedAddress = byte;
          *state = A_RCV;
        }
        else *state = START_;
        break;

      case A_RCV:
        if (DEBUG == 1) fprintf(stdout,"Entered in A_RCV | ");
        if (byte == repeatedCtrl) isRepeated = 1;
        if (byte == FLAG) *state = FLAG_RCV;
        else if (byte == ctrl || isRepeated) {
          receivedControl = byte;
          calculatedBCC = receivedAddress ^ receivedControl;
          *state = C_RCV;
        }
        else *state = START_;
        break;

      case C_RCV:
        if (DEBUG == 1) fprintf(stdout,"Entered in C_RCV | ");
        if (byte == FLAG) *state = FLAG_RCV;
        else if (calculatedBCC == byte) {
          *state = BCC_OK;
          currentDataIdx = 0;
        }
        else *state = START_;
        break;

      case BCC_OK:
        if (DEBUG == 1) fprintf(stdout,"Entered in BCC_OK\n");
        if (currentDataIdx >= STUFF_DATA_MAX) *state = START_;
        else if (byte == FLAG) {
          dataSize = dataDeStuffing(stuffed_data, currentDataIdx, buffer, &bcc2);
          if( createBCC2(buffer, dataSize, &calculatedBCC2) == ERROR ){
            fprintf(stderr,"Error generating BCC2, data field problem\n");
            return ERROR;
          }
          /*Slide 15*/
          if (isRepeated) {
            if (sendSupervisionFrame(fd, A_SR, C_RR(1 - r)) == ERROR) {
              fprintf(stderr, "Error sending supervision frame repeting frame\n");
              return -1;
            }
            *state = START_;
          }
          else if (calculatedBCC2 != bcc2) {
            if (sendSupervisionFrame(fd, A_SR, C_REJ(1 - r)) == ERROR) {
              fprintf(stderr, "Error sending supervision frame rejecting frame\n");
              return ERROR;
            }
            *state = START_;
          }
          else *state = STOP_;
        }
        else stuffed_data[currentDataIdx++] = byte;
        break;

      default:
        break;
    }
  }

  return dataSize;
}
```

```c
int receiverDisc(int fd){
  MACHINE_STATE receiverState = START_;

  if(DEBUG) fprintf(stdout,"Receiving DISC\n");
  while( receiverState != STOP_ ) {
    if( checkSupervisionFrame(&receiverState, fd, A_SR, C_DISC, NULL) == ERROR) return ERROR;
  }

  if( sendSupervisionFrame(fd, A_SR, C_DISC) == ERROR) return ERROR;

  if(DEBUG) fprintf(stdout,"Receiving UA\n");
  receiverState = START_;
  while( receiverState != STOP_ ) {
    if( checkSupervisionFrame(&receiverState, fd, A_SR, C_UA, NULL) == ERROR) return ERROR;
  }

  return SUCCESS;
}


int dataDeStuffing(char *stuffedBuffer, int stuffedBufferSize, char *buffer, char *BCC2){
  char destuffedBuffer[MAX_DATA_SIZE+1]; /* +1 because of trailing '\0' char */

  int temp = 0; /* Auxiliary int */

  for( int i = 0; i < stuffedBufferSize; i++){
    if( stuffedBuffer[i] == ESCAPE ){
      char nextByte = stuffedBuffer[++i];
      if( nextByte == FLAG_ESCAPE_XOR ) destuffedBuffer[temp++] = FLAG;
      else if( nextByte == ESCAPE_XOR ) destuffedBuffer[temp++] = ESCAPE;
      else {
        fprintf(stderr,"Escape byte violation\n");
        return ERROR;
      }
    }
    else destuffedBuffer[temp++] = stuffedBuffer[i];
  }


  *BCC2 = destuffedBuffer[--temp];

  for(int j = 0; j < temp; j++ ) buffer[j] = destuffedBuffer[j];

  return temp;
}
```

```c
int receivedStuffedData(int fd, char *buffer){
  MACHINE_STATE state = START_;

  char stuffedBuffer[STUFF_DATA_MAX];

  int size = receivedStuffedDataSM(&state, fd, stuffedBuffer, buffer);
  if( size == ERROR ){
    fprintf(stderr, "Error verifying stuffed buffer integrity\n");
    return ERROR;
  }

  if( sendSupervisionFrame(fd, A_SR, C_RR(r)) == ERROR ){
    fprintf(stderr, "Error sending supervision frame for all data received well\n");
    return ERROR;
  }

  r = 1 - r;

  return size;
}
```

receiver.h

```c
#ifndef RECEIVER_H
#define RECEIVER_H

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#include "common.h"
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

#define BAUDRATE B38400
#define _POSIX_SOURCE 1 /* POSIX compliant source */
#define FALSE 0
#define TRUE 1

/**
 * @brief Opens the receptor for reading and sending data
 *
 * @param  filename           serial port file
 * @return        File descriptor of the serial port upon sucess, ERROR otherwise
 */
int openReceiver(char filename[]);

/**
 * @brief Acknowledge SET frame, transmitted by the sender, validate, and then
 * send the UA response (slide 7 && 14)
 *
 * @param  fd                 serial port file descriptor
 * @return    SUCCESS upon success, ERROR otherwise
 */
int receiveSetFrame(int fd);

/**
 * @brief Closes the Receiver side of the serial port (slide 14)
 *
 * @param  fd                 serial port file descriptor
 * @return    SUCCESS upon success, ERROR otherwise
 */
int closeReceiver(int fd);

/**
 * @brief Receiver acknowledges DISC frame, validates, resend the DISC frame,
 * and if the DISC frame is validated by the other side of the serial port,
 * acknowledges a UA frame, and consequently validates the frame (slide 14)
 *
 * @param  fd                 serial port file descriptor
 * @return    SUCCESS upon success, ERROR otherwise
 */
int receiverDisc(int fd);

/**
 * @brief Data Destuffing
 * @param  stuffedBuffer        stuffed buffer received from sender
 * @param  stuffedBufferSize    size of the stuffed buffer
 * @param  buffer               buffer after destuffing
 * @param  BCC2                 data BCC2
 * @return     destuffed buffer upon sucess, ERROR otherwise
 */
int dataDeStuffing(char *stuffedBuffer, int stuffedBufferSize, char *buffer, char *BCC2);

/**
 * @brief Receives stuffed data from sender
 * @param  fd                    file descriptor
 * @param  stuffedData           buffer containing stuffed data
 * @return            buffer size upon success, ERROR otherwise
 */
int receivedStuffedData(int fd, char *buffer);

/**
 * @brief State machine for receiving stuffed data frame
 * @param  state                state machine state
 * @param  fd                   serial port file descriptor
 * @param  stuffedBuffer        stuffed buffer received from sender
 * @param  buffer               destuffed buffer
 * @return              buffer size upon success, ERROR otherwise
 */
int receivedStuffedDataSM(MACHINE_STATE *state, int fd, char *stuffedBuffer, char *buffer);



#endif
```

serial.c

```c
#include "serial.h"


int* parseArgs(int argc, char **argv, char *id, char *file, char *serialPort){
  int *flagFrame; /* Debug, Help, Identity ... */
  flagFrame = malloc(4* sizeof *flagFrame);
  int index, c;

  opterr = 0;

  while ((c = getopt (argc, argv, "dhi:f:")) != -1)
    switch (c){
      case 'd':
        flagFrame[0] = 1;
        break;
      case 'h':
        flagFrame[1] = 1;
        break;
      case 'i':
        flagFrame[2] = 1;
        strcpy(id,optarg);
        break;
      case 'f':
        flagFrame[3] = 1;
        strcpy(file,optarg);
        break;
      case '?':
        if (optopt == 'c') fprintf(stderr, "Option -%c requires an argument.\n", optopt);
        else if (isprint (optopt)) fprintf(stderr, "Unknown option `-%c'.\n", optopt);
        else fprintf(stderr,"Unknown option character `\\x%x'.\n",optopt);
        return NULL;
      default:
        abort();
      }

  if(argv[optind] == NULL && (flagFrame[1] != 1) ) {
    printf("here\n");
    return NULL;
  }
  if(argv[optind] != NULL) strcpy(serialPort, argv[optind]);

  return flagFrame;
}

void printHelpMessage(){
  fprintf(stdout, "Usage: serialport [ID]... [OPTION] ... [FILE]...\n");
  fprintf(stdout, "Establish serial port connection to send [FILE] or receive [FILE]\n\n");
  fprintf(stdout, "If file path is not given, pinguim.gif will be sent/received by default, if available on execution root folder\n\n");
  fprintf(stdout, "\t-d\t\t\t enable debug mode, output written to files stdout.txt stderr.txt, respectively\n");
  fprintf(stdout, "\t-i\t\t\t serial port side identifier, sender || receiver\n" );
  fprintf(stdout, "\t-h\t\t\t serial port help message\n");
  fprintf(stdout, "\t-f\t\t\t file to sent/received through the serial port, being ./pinguim.gif the default value if no arguments are provided\n\n");
  fprintf(stdout, "Examples:\n");
  fprintf(stdout, "\tserialport -i sender -d -f ./file.txt\t\t\t Serial port sender side, debug mode active sending file.txt\n");
  fprintf(stdout, "\tserialport -d -i receiver\t\t\t Serial port receiver side, debug mode active receiving pinguim.gif\n");
}

int validateArgs(int argc, char **argv, int *id, char *file, char *identifier, char *serialPort ){
  int *res = parseArgs(argc, argv, identifier,file,serialPort);
  if( res == NULL ){
    fprintf(stderr,"No matching call\n");
    return ERROR;
  }
  if( res[1] == 1 ){
    printHelpMessage();
    return 3;
  }
  if( res[2] == 1 ){
    *id = -1;
    if( (strcmp(identifier,"emissor") != 0) && (strcmp(identifier,"receiver") != 0) ) return -1;
    else {
      *id = (strcmp(identifier,"emissor") == 0) ? 0 : 1;
    }
  }
  if( res[3] != 1 )
  {
    strcpy(file,"./pinguim.gif");
  }
  if( res[0] == 1 ) {
    fprintf(stdout,"Debug mode enabled\n");
  }
  return SUCCESS;
}
```

```c
int execution(int argc, char **argv){
  int *i = malloc(sizeof(int));
  char *id = (char *) malloc(15*sizeof(char));
  char *file = (char *) malloc(100*sizeof(char));
  char *serialPort = (char*) malloc(20*sizeof(char));
  int res = validateArgs(argc, argv, i, file, id, serialPort);
  if( res == ERROR ){
    fprintf(stdout,"Bad input provided, try using serialport -h\n");
    return ERROR;
  }
  else if( res == 3 ) return SUCCESS;

   if ( ((strcmp("/dev/ttyS10", serialPort)) != 0) && ((strcmp("/dev/ttyS11", serialPort)) != 0) ) {

    printf("Usage: application serialPort type <fileName>\nex: application /dev/ttyS0 0 ./imagem.gif\n<fileName> default value: ./pinguim.gif\n");
    exit(1);
  }


  int fd, status = *i;

  /* Sending console stderr and stdout to file When debug is enabled*/
  if(status == SENDERID && DEBUG){
    freopen("Stdout_sender","w",stdout); freopen("Stderr_sender","w",stderr);
  }
  else if( status == RECEIVERID && DEBUG){
    freopen("Stdout_receiver","w",stdout); freopen("Stderr_receiver","w",stderr);
  }

  if (status != SENDERID && status != RECEIVERID) {
    printf("application 2nd argument(type) should be either 0 or 1.\n0-Sender\n1-Receiver\n");
    return ERROR;
  }

  if ((fd = llopen(serialPort, status)) < 0) {
    fprintf(stderr, "Error on llopen\n");
    return ERROR;
  }

  if (status == SENDERID) {
    if (sendFile(fd, file) < 0) {
      fprintf(stderr, "Error sending file\n");
      return ERROR;
    }

  }
  else {
    if (readFile(fd) == ERROR) {
      fprintf(stderr, "Error reading file\n");
      return ERROR;
    }
  }

  if (llclose(fd,status) == ERROR) {
    fprintf(stderr, "Error on llclose\n");
    return ERROR;
  }

  return SUCCESS;

}

int main(int argc, char** argv) {
  return execution(argc,argv);
}
```

serial.h

```c
#ifndef SERIAL_H
#define SERIAL_H

#include "dataLayer.h"
#include "application.h"
#include <stdio.h>
#include <unistd.h>
#include <stdbool.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>


#define serialPort1 "/dev/ttyS10"
#define serialPort2 "/dev/ttyS11"

/**
 * @brief Function to parse arguments and options, given by the terminal
 *
 * @param argc              terminal number of arguments
 * @param argv              terminal arguments
 * @param id                1->Receiver 0->Sender
 * @param file              file being received/sent by the serial port
 * @param serialPort        serial port
 * @return int*             Flag frame, with options disabled/enabled
 */
int* parseArgs(int argc, char **argv, char *id, char *file, char *serialPort);

/**
 * @brief Function to print to stdout serialport usage and running mode
 *
 */
void printHelpMessage();


/**
 * @brief Program flow control
 *
 * @param argc              terminal number of arguments
 * @param argv              terminal arguments
 * @return int              SUCCESS upon success, ERROR otherwise
 */
int execution(int argc, char **argv);

/**
 * @brief Validate the inputs given in terminal by the user
 *
 * @param argc              terminal number of arguments
 * @param argv              terminal arguments
 * @param id                1->Receiver 0->Sender
 * @param file              file being received/sent by the serial port
 * @param identifier        Emissor || Receiver inputs from terminal
 * @param serialPort        serial port
 * @return int              SUCCESS upon success, ERROR otherwise
 */
int validateArgs(int argc, char **argv, int *id, char *file, char *identifier, char *serialPort );;



#endif
```

dataLayer.c

```c
#include "dataLayer.h"

int llopen(char *filename, int id){

  int fd;

  if( id == RECEIVERID ) fd = openReceiver(filename);
  else if( id == SENDERID ) fd = openSender(filename);
  else{
    fprintf(stderr, "Invalid id given, either (0) for sender or (1) for receiver\n");
    return ERROR;
  }

  return fd;
}


int llclose(int fd, int id){
  int result;

  if( id == RECEIVERID ) result = closeReceiver(fd);
  else if( id == SENDERID ) result = closeSender(fd);
  else{
    fprintf(stderr, "Invalid id given, either (0) for sender or (1) for receiver\n");
    return ERROR;
  }

  if( result == ERROR ){
    fprintf(stderr, "Error closing %s", id == 1 ? "receiver" : "sender");
    return ERROR;
  }

  fprintf(stdout,"Closing %s\n", id == 1 ? "Receiver" : "Sender");
  return SUCCESS;
}


int llwrite(int fd, char *buffer, int size, int id){
  if( id == RECEIVERID ){
    fprintf(stderr,"The receiver is unable send data\n");
    return ERROR;
  }
  return sendStuffedFrame(fd, buffer, size);
}


int llread(int fd, char *buffer, int id){
  if( id == SENDERID ){
    fprintf(stderr, "The sender is unable to read data\n");
    return ERROR;
  }

  return receivedStuffedData(fd, buffer);
}
```

dataLayer.h

```h
#ifndef DATALAYER_H
#define DATALAYER_H

#include "sender.h"
#include "receiver.h"


/**
 * @brief Open serial port connection through a file descriptor, in a specific
 * side of the serial port, given by the id
 *
 * @param  filename              file path to be transmmited
 * @param  id                    Receiver -> 1 | Sender -> 0
 * @return          file descriptor upon sucess, ERROR otherwise
 */
int llopen(char *filename, int id);

/**
 * @brief Closes the serial port connection, in a specific
 * side of the serial port, given by the id
 *
 * @param  fd               File descriptor of filename(llopen) file
 * @param  id               Receiver -> 1 | Sender -> 0
 * @return          SUCCESS upon success, ERROR otherwise
 */
int llclose(int fd, int id);

/**
 * @brief Sends a packet with the data , the data can't extend the specified max length
 *
 * @param  fd                   File descriptor of the RS-232 port
 * @param  buffer               Data to be written to serial port
 * @param  size                 Buffer size
 * @param  id                   Receiver -> 1 | Sender -> 0
 * @return        Data size upon sucess, ERROR otherwise
 */
int llwrite(int fd, char *buffer, int size, int id);

/**
 * @brief Receives packet with the data, sent throw the serial port
 *
 * @param  fd                   File descriptor of the RS-232 serial port
 * @param  buffer               Buffer with data read from the serial port
 * @param  id                   Receiver -> 1 | Sender -> 0
 * @return        Data size upon sucess, ERROR otherwise
 */
int llread(int fd, char *buffer, int id);



#endif
```

application.c

```c
int sendFile(int fd, char fPath[]) {

  FILE *fd1;

  if ((fd1 = fopen(fPath, "rb")) == NULL) {
    fprintf(stderr, "Open file failed\n");
    return -1;
  }

  fseek(fd1, 0, SEEK_END);

  long fSize = ftell(fd1);
  rewind(fd1);

  if (sendControlPacket(fd, C_START, fSize, basename(fPath)) < 0){
    return -1;
  }

  if (sendData(fd, fd1, fSize) < 0) {
    return -1;
  }

  if (sendControlPacket(fd, C_END, fSize, basename(fPath)) < 0){
    return -1;
  }

  fclose(fd1);
  return fSize;
}

int sendControlPacket (int fd, u_int8_t cField, long fSize, char fName[]) {

  size_t fNSize = strlen(fName) + 1;

  if (fNSize > 0xff) {
    fprintf(stderr, "File is too big\n");
    return -1;
  }

  size_t pSize =  fNSize + sizeof(long) + 5;
  u_int8_t *controlPacket = malloc(pSize);

  controlPacket[0] = cField;
  controlPacket[1] = T_TYPE_SIZE;
  controlPacket[2] = (u_int8_t) sizeof(long);
  memcpy(controlPacket + 3, &fSize, sizeof(long));
  controlPacket[3 + sizeof(long)] = T_TYPE_NAME;
  controlPacket[4 + sizeof(long)] = (u_int8_t) fNSize;
  memcpy(controlPacket + 5 + sizeof(long), fName, fNSize);

  if (llwrite(fd, controlPacket, pSize, SENDERID) < 0) return -1;
  free(controlPacket);

  return 1;
```

```
  return 1;
}

int sendData(int fd, FILE* fd1, long fSize) {

  u_int8_t* data = malloc(fSize);
  fread(data, sizeof(u_int8_t), fSize, fd1);

  u_int8_t sequence = 0;
  for (int i = 0; i < fSize; i += D_REAL_SIZE) {

    int fDataSize = min(D_REAL_SIZE, fSize - i);
    u_int8_t* fData = malloc(fDataSize);
    memcpy(fData, data + i, fDataSize);

    int dPacketSize = fDataSize + ADD_FIELDS;
    u_int8_t* dPacket = malloc(dPacketSize);
    dataPacketBuilder(dPacket,dPacketSize, fData, fDataSize, sequence);

    if (llwrite(fd, dPacket, dPacketSize, SENDERID) < 0) {
      fprintf(stderr, "Error sending data packet\n");
      return -1;
    }

    free(fData);
    free(dPacket);

    sequence = (sequence + 1) % 256;
  }

  free(data);
  return 1;
}

void dataPacketBuilder(u_int8_t* dPacket,int dPacketSize, u_int8_t* fData, int fDataSize, u_int8_t sequence) {

  u_int8_t l1 = fDataSize % N_MULT;
  u_int8_t l2 = fDataSize / N_MULT;

  dPacket[0] = C_DATA;
  dPacket[1] = sequence;
  dPacket[2] = l2;
  dPacket[3] = l1;

  memcpy(dPacket + ADD_FIELDS, fData, fDataSize);

}
```

```c
int readControlPacket(int fd, u_int8_t cField, u_int8_t buf[], char** fName, long* fSize) {
  int size;

  if ((size = llread(fd, buf, RECEIVERID)) < 0) {
    fprintf(stderr, "Failed reading cPacket\n");
    return -1;
  }

  if (buf[0] != cField) {
    fprintf(stderr, "Error in cField\n");
    return -1;
  }

  int i = 1;
  u_int8_t type, length;
  while (i < size) {
    type = buf[i++];
    length = buf[i++];

    if(type == T_TYPE_SIZE){
      memcpy(fSize, buf+i, length);

    }
    else if(type == T_TYPE_NAME){
      *fName = malloc(length);
        memcpy(*fName, buf+i, length);
    }

    i += length;
  }

  return 1;
}
```

```c
int readFile(int fd) {
  u_int8_t buf[MAX_DATA_SIZE];
  char* fName = NULL;
  long fSize;

  if (readControlPacket(fd, C_START, buf, &fName, &fSize) < 0){
    return -1;
  }

  FILE *fd1;
  if ((fd1 = fopen(fName, "wb")) == NULL) {
    fprintf(stderr, "Open file failed\n");
    return -1;
  }

  int size;
  u_int8_t sequence = 0;

  while ((size = llread(fd, buf, RECEIVERID)) != ERROR) {

    if (buf[0] == C_END) {
      break;
    }
    if (buf[0] != C_DATA) {
      fprintf(stderr, "data packet control byte read failed\n");
      return -1;
    }
    if (buf[1] != sequence) {
      fprintf(stderr, "sequence number in data packet error\n");
      return -1;
    }

    int dataFieldOctets = N_MULT * buf[2]   + buf[3];

    u_int8_t* dField = malloc(dataFieldOctets);

    memcpy(dField, buf + ADD_FIELDS, dataFieldOctets);

    fwrite(dField, sizeof(u_int8_t), dataFieldOctets, fd1);

    sequence = (sequence + 1) % 256;
    free(dField);
  }

  free(fName);
  fclose(fd1);
  if( size == ERROR ) return ERROR;
  return fSize;
}
```

application.h

```h
#define PACKET_DATA_SIZE 200

// T BYTE
#define T_TYPE_SIZE 0
#define T_TYPE_NAME 1

// PACKETS CONSTANTS
#define C_DATA 1
#define C_START 2
#define C_END 3

// DATA PACKET CONSTANTS
#define ADD_FIELDS 4
#define D_REAL_SIZE (PACKET_DATA_SIZE - ADD_FIELDS)
#define N_MULT 256

#define min(a,b) (((a) < (b)) ? (a) : (b))




//

// T BYTE
#define T_FILE_SIZE 0
#define T_FILE_NAME 1


// PACKETS CONSTANTS
#define DATA_CTRL 1
#define START_CTRL 2
#define END_CTRL 3

// DATA PACKET CONSTANTS
#define NUM_DATA_ADDITIONAL_FIELDS 4
#define DATA_ACTUAL_SIZE (PACKET_DATA_SIZE - NUM_DATA_ADDITIONAL_FIELDS)
#define NUM_OCTETS_MULTIPLIER 256

#define min(a,b) (((a) < (b)) ? (a) : (b))


/**
* @brief Builds data packet, from a given data buffer
*/
void dataPacketBuilder(u_int8_t* dPacket,int dPacketSize, u_int8_t* fData, int fDataSize, u_int8_t sequence);
```

```c
/**
* @brief Sends data packet
*
* @param fd                        File Descriptor
* @param fd1                       Pointer to file being trasnferred
* @param fSize                     file size
* @return              1 upon sucess, otherwise -1
*/
int sendData(int fd, FILE* fd1, long fSize);


/**
 * @brief creates and sends a control packet
 *
 * @param  fd                      file descriptor
 * @param  cField                  control field
 * @param  fSize                   file size
 * @param  fName                   file name
 * @return              1 upon sucess, otherwise -1
 */
int sendControlPacket (int fd, u_int8_t cField, long fSize, char fName[]);

/**
 * @brief Sends the controlPackets and the data packet
 *
 * @param  fd                      File descriptor
 * @param  fPath                   File Path
 * @return          file Size upon success, otherwise -1
 */
int sendFile(int fd, char fPath[]);


/**
 * @brief Reads all packets and creates the file read
 *
 * @param  fd                      File descriptor
 * @return    file Size upon success, otherwise -1
 */
int readFile(int fd);

/**
 * @brief reads a control packet and alocates the file name and file size
 *
 * @param  fd                      file descriptor
 * @param  cField                  control field
 * @param  buf                     buffer where the control packet is
 * @param  fName                   file name
 * @param  fSize                   file size
 * @return              1 upon sucess, otherwise -1
 */
int readControlPacket(int fd, u_int8_t cField, u_int8_t buf[], char** fName, long* fSize);
```

common.c

```c
#include "common.h"

int isRejected;

int checkSupervisionFrame(MACHINE_STATE *state, int fd, char A_BYTE, char C_BYTE, char* reject){
  char frame_byte;

  if( getBytefromFd(fd, &frame_byte) == ERROR ) return ERROR;

  switch(*state){
    case START_:
      if (DEBUG == 1) fprintf(stdout,"Entered in START_ | ");
      if( frame_byte == FLAG ) *state = FLAG_RCV;
      break;
    case FLAG_RCV:
      isRejected = 0;
      if (DEBUG == 1) fprintf(stdout,"Entered in FLAG_RCV | ");
      if( frame_byte == FLAG ) return SUCCESS;
      else if( frame_byte == A_BYTE ) *state = A_RCV;
      else *state = START_;
      break;
    case A_RCV:
      if (DEBUG == 1) fprintf(stdout,"Entered in A_RCV | ");
      if( reject != NULL && frame_byte == *reject ) {
        isRejected = 1;
      }
      if( frame_byte == FLAG ) *state = FLAG_RCV;
      else if( frame_byte == C_BYTE || isRejected ) *state = C_RCV;
      else *state = START_;
      break;
    case C_RCV:
      if (DEBUG == 1) fprintf(stdout,"Entered in C_RCV | ");
      if( frame_byte == FLAG ) *state = FLAG_RCV;
      else if( frame_byte == BCC(A_BYTE,C_BYTE)  ) *state = BCC_OK;
      else *state = START_;
      break;
    case BCC_OK:
      if (DEBUG == 1) fprintf(stdout,"Entered in BCC_OK\n");
      if( frame_byte == FLAG ) *state = STOP_;
      else *state = START_;
      break;
    default:
      if (DEBUG == 1) fprintf(stdout,"Default statement reached\n");
      break;
    }
    if( *state == STOP_ && isRejected ){
      *state = START_;
      return 1;
    }
    return SUCCESS;
}


int getBytefromFd(int fd, char *byte_to_be_read){
  int res = read(fd, byte_to_be_read, 1);
  if( res == ERROR){
    fprintf(stderr,"Error reading byte from fd\n");
    return ERROR;
  }
  if(DEBUG) fprintf(stdout,"Byte read: %02x , ", *byte_to_be_read);
  return SUCCESS;
}


int sendSupervisionFrame(int fd, char A_BYTE, char C_BYTE){
  char frame[5] = {FLAG, A_BYTE, C_BYTE, BCC(A_BYTE,C_BYTE), FLAG};

  int res = write(fd, frame, 5);

  if( res == ERROR ){
    fprintf(stderr,"Error writing to serial Port\n");
    return ERROR;
  }
  return SUCCESS;
}

int createBCC2(char *buffer, int bufferSize, char *bcc2){
  if(sizeof(buffer) > 0){
    char BCC2 = buffer[0];

    for(int i = 1; i < bufferSize; i++){
      BCC2 ^= buffer[i];
    }

    *bcc2 = BCC2;
    return SUCCESS;
  }
  else{
    fprintf(stderr,"Unable to create BCC2, buffer not allocated correctly\n");
    return ERROR;
  }
}

void insertError(char *data, int size, int probability){
  int r = rand() % 100;

  if( r < probability ) data[size] += 2;
}
```

common.h

```c
#ifndef STATEMACHINE_H
#define STATEMACHINE_H

#include "macros.h"
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

/*! State machine enumeration class */
typedef enum {
   START_,        /* Start state */
   FLAG_RCV,      /* 0x7E is validated */
   A_RCV,         /* A Field byte is validated*/
   C_RCV,         /* C Field byte is validated */
   BCC_OK,        /* BCC(A_FIELD, C_FIELD) byte is validated */
   STOP_          /* Flags validated correctly */
 } MACHINE_STATE;


/**
 * @brief State Machine for checking I/S/U Frame responses
 *
 * @param  state               the current SM state, passed by reference so it can be updated
 * @param  fd                  serial port file descriptor
 * @param  A_BYTE              A field of I/S/U Frame
 * @param  C_BYTE              A field of I/S/U Frame
 * @param  reject              Negative Acknowledgement
 * @return        SUCCESS upon success, ERROR otherwise
 */
int checkSupervisionFrame(MACHINE_STATE *state, int fd, char A_BYTE, char C_BYTE, char* reject);

/**
 * @brief Read the serial port file descriptor byte by byte. Helper function in the
 * project state machine, to verify and validade answers from one side to the otherwise
 *
 * @param  fd                          serial port file descriptor
 * @param  byte_to_be_read             byte read from the serial port
 * @return                SUCCESS upon success, ERROR otherwise
 */
int getBytefromFd(int fd, char *byte_to_be_read);

/**
 * @brief Send UA, DISC, SET Frames from one side to the other of the serial port
 * @param  fd                  serial port file descriptor
 * @param  A_BYTE              A Field of the UA/DISC/SET Frame
 * @param  C_BYTE              C Field of the UA/DISC/SET Frame
 * @return        SUCCESS upon success, ERROR otherwise
 */
int sendSupervisionFrame(int fd, char A_BYTE, char C_BYTE);

/**
 * @brief Create Information Trame BCC2, consisisting in a xor of all the data field
 * bits, therefore creating BCC2
 *
 * @param  buffer                 Information camp
 * @param  bufferSize             Information camp size
 * @return           BCC2 upon success, '\0' otherwise
 */
int createBCC2(char *buffer, int bufferSize, char *bcc2);

/**
 * @brief Create errors at runtime, to ensure the implementation strength
 *
 * @param data                    Buffer
 * @param size                    Index of data buffer that will be changed
 * @param probability             Probability of introducing error
 */
void insertError(char *data, int size, int probability);

#endif
```

macros.h

```c
#ifndef MACROS_H
#define MACROS_H

#define ALARM_INTERVAL 3                        /* Time interval between triggering the signal */
#define MAX_NO_ANSWER 5                         /* Max number of timeouts accepted */

#define SENDERID 0
#define RECEIVERID 1

#define DEBUG 1                                 /* Debug flag */

#define BIT(n) 1 << n                           /* Bit mask */

#define FLAG 0x7E                               /* Start and Stop trame byte */
#define A_SR 0x03                               /* A byte for Sender to Receiver */
#define A_RS 0x01                               /* A byte for Receiver to sender */
#define C_UA 0x07                               /* Ua Trame Cfield */
#define C_SET 0x03                              /* Set Trame CField */
#define C_DISC 0x0B                             /* Disc Trame CField */
#define C_FRAME_I(n) ( BIT(6*(n)) & 0x40)       /* I frame stuffing logical and. 0x40 in case s->1 0x00 in case s->0 */
#define C_RR(n) (BIT(7*(n)) | 0x05)             /* S/U Frame RR field logical and 0x85 in case n->1 0x05 in case n->0 */
#define C_REJ(n) (BIT(7*(n)) | 0x01)            /* S/U Frame REJ field logical and 0x81 in case n->1 0x01 in case n->0 */
#define ESCAPE 0x7D                             /* Escape Byte */
#define FLAG_ESCAPE_XOR 0x5E                    /* FLAG ^ 0x20 */
#define ESCAPE_XOR 0x5D                         /* Escape ^ 0x20 */


#define BCC(a,c) (a ^ c)                        /* Bcc flag is generated by A Byte XOR with C byte */

#define MAX_DATA_SIZE 200
#define STUFF_DATA_MAX ( MAX_DATA_SIZE * 2+ 2 ) /* Data + frameH + BCC2 */

#define ERROR -1                                /* Error return type */
#define SUCCESS 0                               /* Success return type */

#endif
```