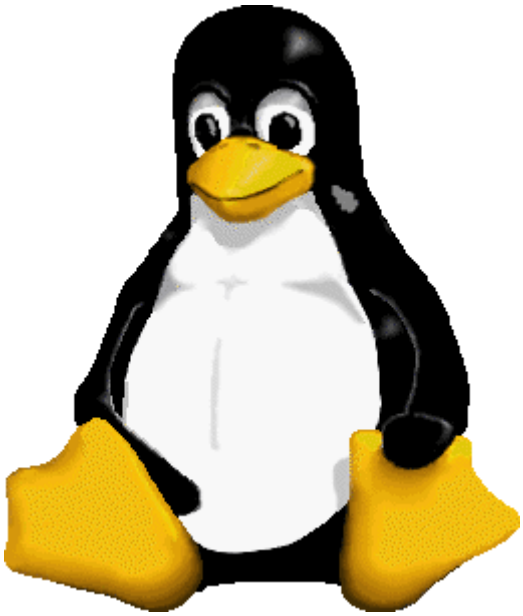
Faculdade de Engenharia da Universidade do Porto



Relatório do 1º trabalho laboratorial de RCOM

Autores:

João Marinho up201905952@fe.up.pt

Tiago Silva up201906045@fe.up.pt

ÍNDICE

SUMÁRIO

No âmbito da unidade curricular de Redes de Computadores (RC), desenvolvemos um protocolo de ligação de dados como 1º trabalho laboratorial. Este protocolo, baseia-se numa especificação dada pelos docentes e foi testado com uma aplicação simples, igualmente especificada, de transferência de ficheiros.

Desta forma, o trabalho cumpriu os objetivos estabelecidos podendo até dizer-se que foi finalizado com sucesso, capaz de transferir dados mesmo com a adversidade da existência de ruído ou a interrupção da ligação.

1 INTRODUÇÃO

O principal objetivo deste projeto, passa por levar os alunos a implementarem o seu próprio protocolo de ligação de dados, de forma a fornecer um serviço de comunicação de dados fiável entre dois sistemas ligados por um meio de transmissão – neste caso, um cabo série.

Deste modo, neste relatório, explicamos como é feita a transmissão entre os sistemas – quais os procedimentos, todo o desenrolar da ação, bem como uma especificação e explicação das estruturas do código desenvolvido.

Por fim, explicamos também quais os testes a que o código foi submetido e apresentamos uma caracterização estatística da eficiência do protocolo, de forma a conseguirmos ter uma perceção concreta do desempenho do protocolo desenvolvido.

2 ARQUITETURA

Neste projeto são concentradas três distintas camadas ou blocos funcionais, com os quais tivemos que operar. Estas são: a aplicação, o protocolo de ligação de dados e a porta de série.

A aplicação, camada de mais alto nível, é responsável pela comunicação com o segundo bloco funcional através da *Interface* protocolo-aplicação. Além disso, abstrai o utilizador das outras camadas, isto é, aos olhos do utilizador os pacotes de dados enviados passam da camada de aplicação do PC1 para o PC2 de forma direta, enquanto na verdade existem sucessivos pedidos às camadas inferiores.

O protocolo de ligação de dados possui uma maior responsabilidade no que toca à transmissão de dados, já que o seu objetivo final passa por fornecer um serviço de comunicação fiável entre dois sistemas por meio de um cabo de série (neste caso). Desde controlo de erros, confirmações positivas, e estabelecimento e termino da ligação, são tudo operações que o protocolo de ligação de dados tem de ter em conta.

Finalmente, o meio físico por onde os dados são enviados designa-se por porta de série, esta que possui um driver de forma a poder comunicar com o protocolo de ligação de dados com a ajuda de uma API.

3 ESTRUTURAS DE CÓDIGO

De forma a interligar todas as camadas descritas anteriormente, foi necessário definir uma estruturação consistente e adequada de forma a tornar cada um dos blocos funcionais independentes mantendo uma hierarquia, isto é, cada bloco comunica apenas com o bloco diretamente abaixo ou acima de forma a enviar pedidos ou responder a estes, respetivamente.

No que concerne a aplicação existem duas funções principais – ***appRead***, ***appWrite*** – estas que, como o nome indica são utilizadas pelos dois atores do programa, o *Receiver* e o *Transmitter*. Existe também uma estrutura de dados auxiliar a esta camada – ***applicationLayer*** – cujos parâmeros são o *file descriptor* da porta de série, e o status que nos indica qual o tipo de ator.

No caso do *Transmitter* a função de escrita será executada, esta que irá começar pelo envio de um pacote de controlo indicando o início do envio de pacotes de dados, seguido dos respetivos e que termina com o envio do pacote de controlo de término. Por outro lado, o *Receiver* executa a função de leitura que irá receber tanto os pacotes de controlo e de dados, processando-os devidamente.

A fim de existir uma comunicação entre a camada de aplicação e o protocolo de dados, foram criadas quatro funções distintas, que servem de interface ao lidar com os pedidos da camada superior para a camada inferior, estas são – ***llopen***, ***llclose***, ***llwrite***, ***llread***.

A função que é executada antes de qualquer outra operação, ***llopen***, é responsável pela configuração da porta de série e respetiva ligação entre os dois *PCs* que estão a correr o programa. Como o nome indica, a função ***llwrite*** é usada para fazer pedidos de envio de pacotes de informação, já a ***llread*** tem como objetivo fazer a leitura desses mesmos pacotes, ambas as funções retornam o tamanho de caracteres escritos e lidos, respetivamente, ou um valor negativo em caso de erro. Por fim, a função ***llclose*** trata de fechar a conexão dos dois *PCs* e de dar *reset* às configurações da porta de série.

No que toca a estruturas de dados presentes no bloco funcional do protocolo de dados possuímos a estrutura – ***linkLayer*** – esta que guarda o número de sequência da trama de envio ou de leitura conforme o ator atual, o número de tentativas máximo de leitura de tramas após o recetor não receber uma confirmação por parte do leitor, e por fim o estado da *flag* de alarme que no caso de estar ativa indica a possibilidade de retransmissão da trama de envio.

4 CASOS DE USO PRINCIPAIS

Tal como referido no ponto três, o programa começa por chamar a função ***llopen***, quer do ponto de vista do *Receiver* como do *Transmitter*, embora o comportamento da mesma seja diferente para os dois atores, isto é, no caso do *Transmitter* após a configuração da porta de série é enviada uma trama não numerada do tipo **Set up**, e a espera de uma trama do tipo **Unnumbered acknowledgment**, através da função – ***sendControl*** – cujos argumentos especificam o *byte* de controlo a ser enviado e o de espera. No que concerne o *Receiver* é feito o inverso, ou seja, a espera da trama de **Set up** e de seguida o envio da trama **UA**, através da função – ***receiveControl*** – que recebe os valores dos bytes de controlo a receber e enviar. Ambas as funções descritas utilizam as funções auxiliares – ***sendMessage***, ***receiveMessage*** – cujo

intuito é bastante explícito, no caso da receção das tramas é, também, utilizada uma *state machine* própria para este tipo de tramas - ***stateMachineSU***.

Tendo configurado a porta de série e por sua vez estabelecido a ligação dos dois *PCs* com sucesso, começa a divisão de tarefas de acordo com o ator do programa. Assim, no que diz respeito ao *Transmitter*, são feitas as seguintes chamadas:

- ***appWrite*** começa por avisar o *Receiver* do início de envio de informação com uma chamada a ***llwrite*** com o pacote de controlo do tipo *start.* Dentro desta, é criada a trama respetiva da camada do protocolo de ligação de dados, e protegida com a chamada da função ***stuffing*** antes de ser enviada através da função ***sendData***.
- No que toca à validação do envio das tramas, é utilizada a ***stateMachineSender***, função que serve de *state machine* para a receção de tramas de supervisão como **receiver ready** e **reject**, cujas chamadas são feitas dentro da função **sendData**.
- Concluída a primeira chamada a ***llwrite***, o procedimento seguinte segue o mesmo caminho visto que esta função é chamada inúmeras vezes dentro de um *loop,* na função ***appWrite***, que trata de enviar pacotes de dados com o conteúdo do ficheiro a transmitir.
- Por fim, será feita outra chamada a ***llwrite***, desta vez, de forma a avisar o *Receiver* que chegou o fim do envio de informação, com um pacote de controlo do tipo *end.*

Quanto ao *Receiver*:

- ***appRead*** começa por chamar a função ***llread*** de maneira a saber que os dados do ficheiro a transmitir serão enviados. Esta chama internamente a função ***receiveData*** que utiliza a ***stateMachineReceiver*** de forma a saber quando sair do ciclo de leituras da porta de série, uma vez que as mesmas são feitas byte a byte.
- Uma vez acabada de ler toda a trama, será feito o seu processamento após respetivo *destuffing* com a função ***destuffing***. Existem vários casos possíveis para o estado da trama recebida, desde cabeçalho errado, trama duplicada, erro no BCC2, todos estes são analisados e se necessário o envio de uma trama de supervisão, **receiver ready** ou **reject,** é feito com a chamada à função ***sendMessage***.
- Concluída a primeira chamada a ***llread***, o procedimento seguinte segue o mesmo caminho visto que esta função é chamada inúmeras vezes dentro de um *loop,* na função ***appRead***, que trata de receber pacotes de dados, com o conteúdo do ficheiro a transmitir, e por sua vez os escreve para o novo ficheiro criado.
- Por fim, será feita outra chamada a ***llread***, cuja mensagem a ler irá simbolizar o fim do envio de informação por parte do *Transmitter.*

Assim que os atores terminem o seu propósito de transmissão e leitura de dados, será feita uma chamada à função ***llclose***, de forma a fechar a ligação entre os dois *PCs* e repor a configuração inicial da porta de série. Este fim de ligação será feito com o envio de uma trama **Disc** por parte do *Transmitter*, receção da mesma por parte do *Receiver* que também irá enviar uma trama **Disc** e finalmente o envio de um **UA**, pelo *Transmitter*, para confirmar a receção do último **Disc** enviado. Mais uma vez, são usadas as funções – ***sendControl, sendMessage, receiveControl e receiveMessage*** – para o efeito.

5 PROTOCOLO DE LIGAÇÃO LÓGICA

Quanto ao nível lógico, o protocolo conta com 3 máquinas de estados:

- **stateMachineSU** – máquina de estados utilizada para na receção das mensagens de controlo, que verifica se a estrutura das tramas é a correta.
- **stateMachineSender** – máquina de estados utilizada pelo *Transmitter* na receção de respostas enviadas pelo recetor. Esta máquina de estados verifica a estrutura da resposta e guarda, através do pointer **controlField**, o valor da resposta (**REJ** ou **RR**).
- **stateMachineReceiver** – máquina de estados utilizada pelo *Receiver* na receção de uma trama. Esta máquina de estados verifica se a trama é totalmente enviada, chegando ao estado **STOP** em caso afirmativo, ou se a trama foi interrompida e iniciou-se o reenvio da mesma, no caso de receber duas *flags* seguidas (devido ao *stuffing* existente, só serão enviadas duas *flags* seguidas se uma trama tiver chegado incompleta).

Deste modo, dependendo do conteúdo da trama recebida, o *Receiver* pode enviar diferentes respostas para o *Transmitter* sendo essas:

- **RR** – tanto no caso de receber uma nova trama correta como no caso da trama vir duplicada, o *Receiver* ignora a trama recebida e envia uma resposta a solicitar a próxima trama (**RR** com o próximo sequence number).
- **REJ** – por outro lado, se a nova trama vier com o **BBC2** errado, ou seja, se a operação **xor** entre todos os bytes do campo de dados for diferente ao penúltimo byte da trama, o *Receiver* vai enviar uma mensagem a solicitar o reenvio da mesma.
- No caso do restante cabeçalho chegar errado, a trama é ignorada e o *Receiver* não enviará nenhuma resposta, dado que pela falta da mesma, o *Transmitter* reenviará novamente a mesma trama.

Por fim, caso a mensagem que o *Transmitter* recebe seja de validação, ou seja, a trama foi corretamente enviada, a função **llwrite** retorna a quantidade de caracteres enviados e **llread**, por sua vez, a quantidade de caracteres lidos.

6 PROTOCOLO DE APLICAÇÃO

A nível da aplicação, apenas há o conhecimento da existência das funções:

- **llopen**
- **llread**
- **llwrite**
- **llclose**

onde apenas existe o conhecimento da finalidade delas, sem qualquer perceção do meio que estas seguem para atingir o fim.

Deste modo, a aplicação começa por receber um input do utilizador:

- No caso do utilizador desejar transmitir um ficheiro, deverá utilizar a aplicação da seguinte forma:

./app <serial-port> <file-name> 1

o que indica qual o canal a utilizar, qual o ficheiro a enviar e o identificador que descreve que a aplicação deve ser utilizada como *TRANSMITTER*.

- No caso do utilizador desejar receber um ficheiro, deverá utilizar a aplicação da seguinte forma:

  ./app <serial-port> 0

Indicando qual o canal a utilizar e o identificador que descrever que a aplicação deve ser utilizada como *RECEIVER*.

Assim, reunidas todas as informações necessárias para a transmissão de um sistema para o outro, a aplicação vai começar por criar uma estrutura de dados do tipo ***applicationLayer*** – ***appLayer*** – onde vai especificar qual o seu identificador – ***appLayer.status*** – e onde vai guardar o *file descriptor* – ***appLayer.fileDescriptor*** - recebido aquando da abertura do canal com a chamada da função ***llopen***.

```
//Receive Serial Port and status specification
appLayer.status = atoi(argv[argc - 1]);
appLayer.fileDescriptor = llopen(argv[1], appLayer.status);
```

De seguida, a aplicação segue rumos diferentes, dependendo de qual o identificador dado no momento da sua chamada:

- No caso do *TRANSMITTER*, a aplicação irá executar a função ***appWrite*** – onde irá transmitir o ficheiro (através da função ***llwrite*** do protocolo de ligação de dados) - fechando o canal quando esta terminar (através da chamada da função ***llclose***).
- No caso do *RECEIVER*, a aplicação irá executar a função ***appRead*** – onde irá receber o ficheiro (através da função ***llread*** do protocolo de ligação de dados) -fechando, também, o canal quando esta terminar (através da chamada da função ***llclose***);

```
switch (appLayer.status) {
  case RECEIVER:
    res = appRead(appLayer.fileDescriptor);
    llclose(appLayer.fileDescriptor, RECEIVER);
    break;
  case TRANSMITTER:
    res = appWrite(appLayer.fileDescriptor, argv[2]);
    llclose(appLayer.fileDescriptor, TRANSMITTER);
    break;
}
```

Assim, a função assegura a transferência de ficheiros, testando com sucesso, o protocolo de ligação de dados desenvolvido.

7 VALIDAÇÃO

Com vista a validar o protocolo desenvolvido, foram efetuados alguns testes para pôr à prova o mesmo:

- Envio de ficheiros de diferentes tamanhos
- Interrupção da ligação durante o envio (com reinício segundos depois)
- Envio de ficheiro com a presença de ruído sobre a ligação
- Envio de ficheiro com a variação do tamanho dos pacotes
- Envio com variação na probabilidade de erros simulados

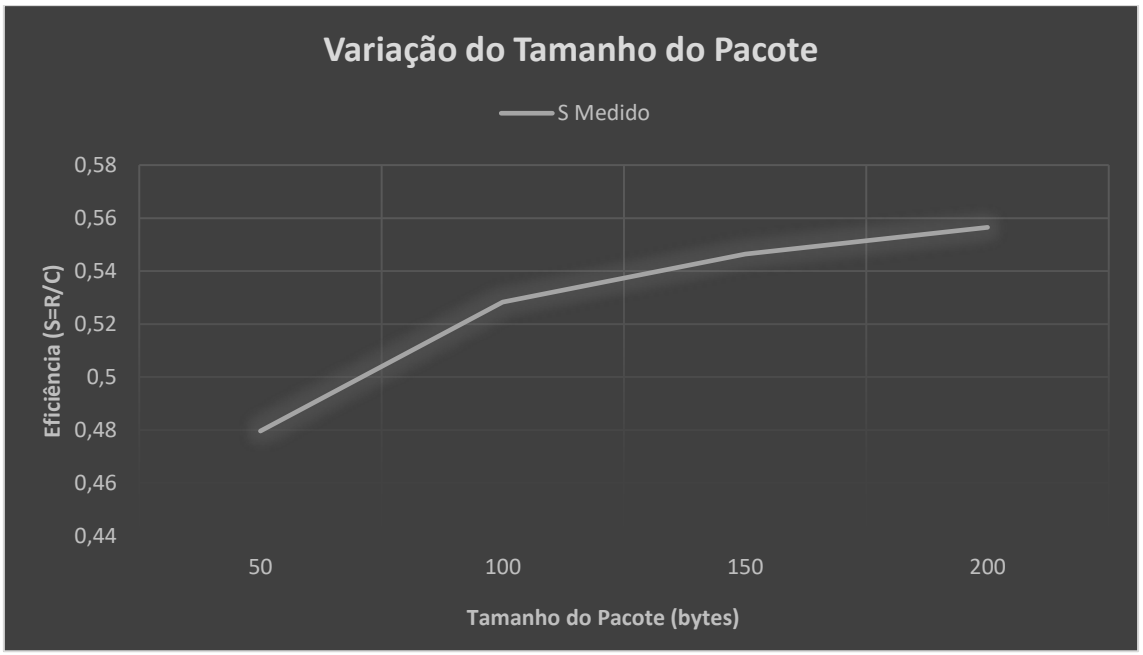
Todos os testes foram concluídos com sucesso.

8 EFICIÊNCIA DO PROTOCOLO DE LIGAÇÃO DE DADOS

De modo a avaliar a eficiência do protocolo desenvolvido, foram testados 3 parâmetros, estes que são: o tamanho dos pacotes enviados, a probabilidade de erro nas tramas (FER) e o valor de tempo de propagação. Para cada um, são apresentados os respetivos gráficos e tabelas presentes no *Anexo II.*

VARIAÇÃO DO TAMANHO DOS PACOTES

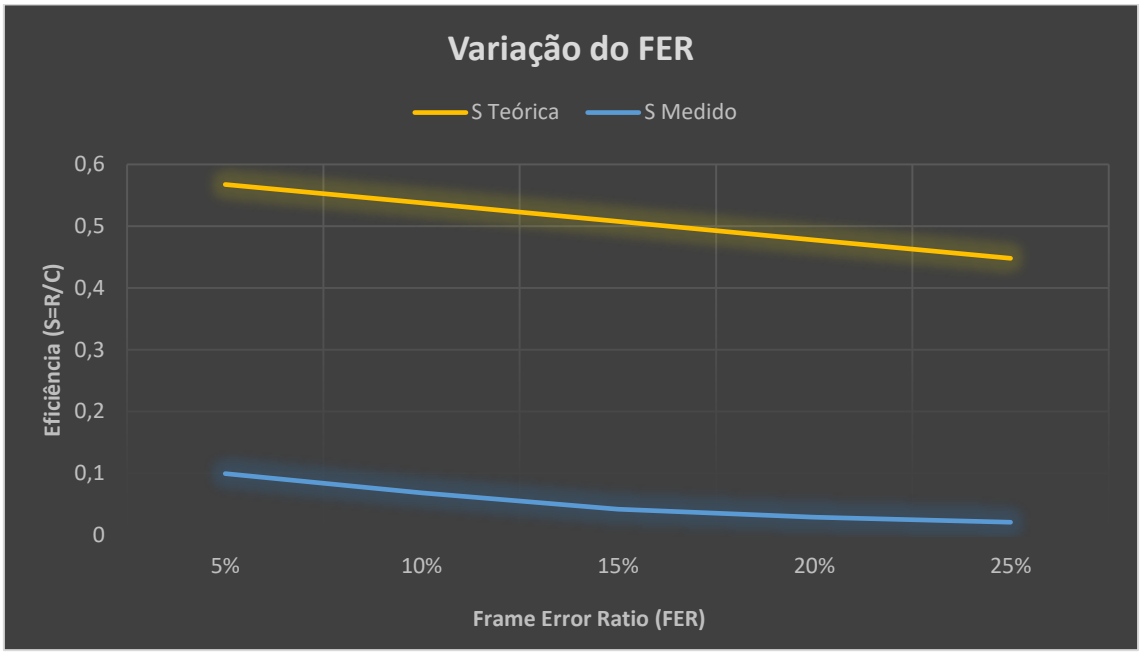
Conseguimos notar um aumento da eficiência com o aumento do tamanho dos pacotes a serem enviados. Isto acontece, pois como são enviados mais dados por pacote, o tempo de transmissão irá ser menor para a mesma quantidade de dados enviados.



VARIAÇÃO DO FER

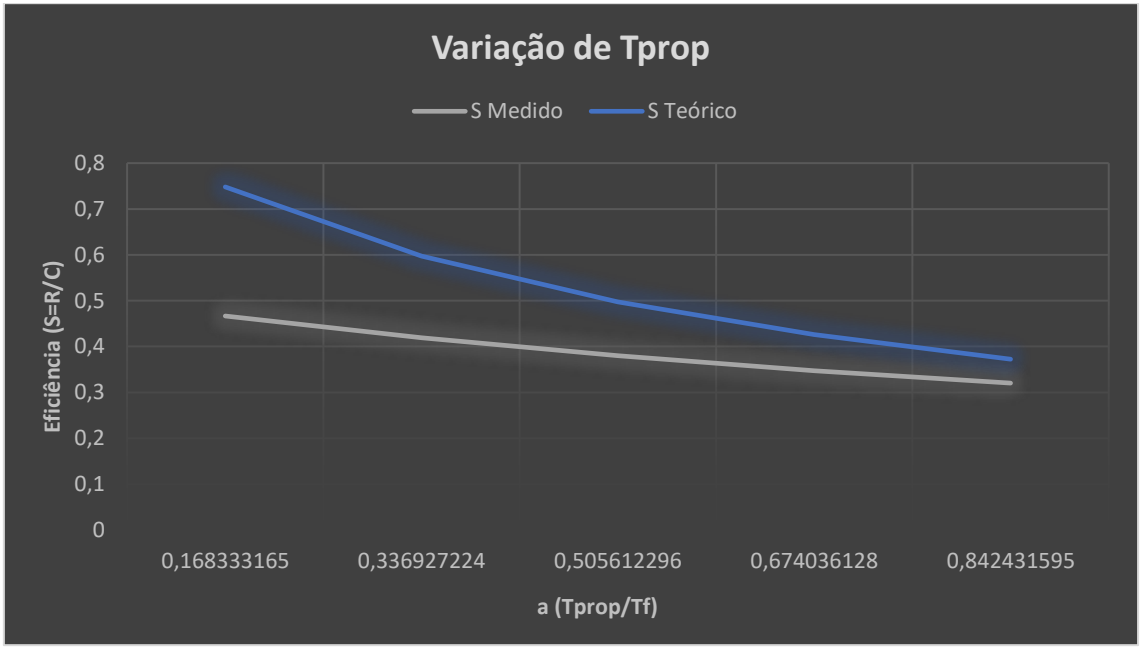
Por outro lado, quando analisamos o gráfico da eficiência pela probabilidade da ocorrência de erro numa trama, o que podemos concluir, é que quanto maior esta probabilidade, menor a eficiência (pois com a ocorrência de mais erros, o protocolo torna-se menos eficiente). Para podermos calcular estes valores utilizamos este pequeno excerto de código que irá gerar um erro com a probabilidade desejada:

```c
/* FER */
int probabilidade = 25;
int number = rand();
if ((number % 100) < probabilidade) {
  printf("%d\n", number);
  sendMessage(fd, A_RCV, REJ(ll.sequenceNumber));
  return 0;
}
```



VARIAÇÃO DO TPROP

Como é possível analisar através do gráfico, percebemos que o valor da eficiência diminui com o aumento de ***Tprop*** e consequentemente o aumento de ***a***. Podemos também verificar pelo gráfico que os resultados obtidos foram bastante positivos pelo que pouco se diferenciam dos valores teóricos.



9 CONCLUSÕES

Com o desenvolvimento deste projeto, colmatamos certos conhecimentos teóricos lecionados nas aulas teóricas. Deste modo, conseguimos ter uma melhor perceção sobre a **independência entre camadas**, pois acabamos por ter de desenvolver diferentes camadas (*application layer* e *link layer*) que em nada dependiam das outras.

Por outro lado, com o desenvolvimento do protocolo, conseguimos compreender alguns dos cuidados que é necessário ter para se conseguir desenvolver um meio que proporciona uma comunicação de dados fiável entre dois sistemas.

Assim, consideramos que este desenvolvimento trouxe bastantes benefícios, pois para além de ter sido uma excelente forma de entrar em contacto com os conteúdos da unidade curricular, também nos ajudou a entender melhor o funcionamento e o conceito de muitos temas desta área.

ANEXO I

application_layer.h:

```
#ifndef _APPLICATION_LAYER_H
#define _APPLICATION_LAYER_H

#include <string.h>
#include <stdio.h>
#include <math.h>
#include <sys/stat.h>
#include "./link_layer.h"

#define PACKET_LENGTH 100

struct applicationLayer {
  int fileDescriptor;
  int status;
};

/**
 * @brief Provides the application layer for the reader
 *
 * @param fd - serial port file descriptor
 * @return int - 0 on success, -1 otherwise
 */
int appRead(int fd);

/**
 * @brief Provides the application layer for the transmitter
 *
 * @param fd - serial port file descriptor
 * @param name - file path, representing the file to transmit
 * @return int - 0 on success, -1 otherwise
 */
int appWrite(int fd, char * name);

int main(int argc, char** argv);

#endif /*_APPLICATION_LAYER_H*/
```

application_layer.c:

```
#include "../headers/application_layer.h"

int appRead(int fd) {

  unsigned char buffer[255];
```

```
  int res, current_index = 0;
  off_t file_length_start = 0, current_position = 0, data_length = 0,
file_length_end = 0;
  unsigned int name_size = 0;

  /* Receive start packet */
  if((res = llread(fd, buffer)) < 0) {
    perror("llread");
    exit(-1);
  }

  if(buffer[current_index++] != START_PKT) {
    perror("Start packet");
    exit(-1);
  }

  /* FILE LENGTH */
  if(buffer[current_index++] != FILE_SIZE) {
    perror("Start- Wrong type");
    exit(-1);
  }
  for(size_t i = 0, index = current_index++; i < buffer[index]; i++) {
    file_length_start = (file_length_start * 256) +
buffer[current_index++];
  }

  /* FILE NAME */
  if(buffer[current_index++] != FILE_NAME) {
    perror("Start- Wrong type");
    exit(-1);
  }

  name_size = buffer[current_index];
  char * name = (char *)malloc(name_size);
  for(size_t i = 0, index = current_index++; i < buffer[index]; i++) {
    name[i] = buffer[current_index++];
  }

  int file_fd;


  if((file_fd = open(name, O_WRONLY | O_CREAT, 0444)) < 0) {
    free(name);
    perror("Open file");
    exit(-1);
  }

  /* Packet processing */
  while(current_position != file_length_start) {
```

```
    if((res = llread(fd, buffer)) < 0) {

      free(name);
      close(file_fd);
      perror("llread");
      exit(-1);
    }

    if(!res) continue;

    current_index = 0;
    if(buffer[current_index++] != DATA_PKT) {

      free(name);
      close(file_fd);
      perror("File not full received");
      exit(-1);
    }

    current_index++;

    data_length = buffer[current_index] * 256 + buffer[current_index+1];

    current_index+=2;
    char * data = (char *)malloc(data_length);
    for(size_t i = 0; i < data_length; i++) {
      data[i] = buffer[current_index++];
    }

    current_position += data_length;

    if(write(file_fd, data, data_length) < 0) {
      free(data);
      free(name);
      close(file_fd);
      perror("Write to file");
      exit(-1);
    }

    free(data);
  }

  close(file_fd);

  /* Receive end packet */
  current_index = 0;
  if((res = llread(fd, buffer)) < 0) {
    free(name);
    perror("llread");
```

```
    exit(-1);
  }

  if(buffer[current_index++] != END_PKT) {
    free(name);
    perror("End packet");
    exit(-1);
  }

  /* FILE LENGTH */
  if(buffer[current_index++] != FILE_SIZE) {
    free(name);
    perror("End- Wrong type");
    exit(-1);
  }

  for(size_t i = 0, index = current_index++; i < buffer[index]; i++) {
    file_length_end = (file_length_end * 256) + buffer[current_index++];
  }

  if(file_length_end != file_length_start) {
    free(name);
    perror("End - Wrong length");
    exit(-1);
  }

  /* FILE NAME */
  if(buffer[current_index++] != FILE_NAME) {
    free(name);
    perror("End - Wrong type");
    exit(-1);
  }

  if(buffer[current_index++] != name_size) {
    free(name);
    perror("End - Wrong name size");
    exit(-1);
  }

  for(size_t i = 0; i < name_size; i++) {
    if(name[i] != buffer[current_index++]) {
      free(name);
      perror("End - Wrong name");
      exit(-1);
    }
  }

  free(name);
  return 0;
```

```
}

int appWrite(int fd, char * name) {
  struct stat fileInfo;
  unsigned int current_index = 0;
  int resW=0, resR = 0, sequence_number = 0;

  if (stat(name, &fileInfo) == -1) {
      perror("stat");
      exit(EXIT_FAILURE);
  }
  off_t length = fileInfo.st_size;
  unsigned char control_packet[255];
  control_packet[current_index++] = START_PKT;
  control_packet[current_index++] = FILE_SIZE;
  control_packet[current_index] = ceil(log2(length) / 8.0);
  unsigned char *length_buf = (unsigned char
*)malloc(control_packet[current_index]);

  for(size_t i = control_packet[current_index++]; i > 0; i--) {
      length_buf[i - 1] = length >> (8*(2-i));
  }

  for(size_t i = 0; i < control_packet[2]; i++) {
    control_packet[current_index++] = length_buf[i];
  }

  control_packet[current_index++] = FILE_NAME;
  control_packet[current_index++] = strlen(name);
  for(size_t i = 0; i < strlen(name); i++) {
    control_packet[current_index++] = name[i];
  }

  /* Send start packet */
  llwrite(fd, control_packet, current_index);

  /* Open transmission file */
  int file_fd;
  if((file_fd = open(name, O_RDONLY)) < 0) {
    free(length_buf);
    perror("Open file");
    exit(-1);
  }

  unsigned char data[PACKET_LENGTH];
  unsigned char *data_packet = (unsigned char *)malloc(PACKET_LENGTH);
  off_t current_position = 0;

  /* Send data packets */
```

```
  while (current_position != length) {

    if((resR = read(file_fd, data, PACKET_LENGTH)) < 0) {
      free(data_packet);
      free(length_buf);
      exit(-1);
    }

    data_packet = (unsigned char *)realloc(data_packet, resR+4);
    data_packet[0]= DATA_PKT;
    data_packet[1]= sequence_number%255;
    data_packet[2]= (resR/256);
    data_packet[3]= resR%256;
    for (size_t i = 0; i < resR; i++)
    {
      data_packet[4+i] = data[i];
    }
    if((resW = llwrite(fd, data_packet, resR + 4)) < 0) {
      free(data_packet);
      free(length_buf);
      exit(-1);
    }

    current_position += resR;
    sequence_number++;
  }

  control_packet[0] = END_PKT;

  /* Send end packet */
  llwrite(fd, control_packet, current_index);
  free(data_packet);
  free(length_buf);

  return 0;
}

int main(int argc, char** argv) {
  struct applicationLayer appLayer;
  int res = 0;

  if((argc != 3 && argc != 4) ||
     (argc == 3 && atoi(argv[argc - 1]) != RECEIVER) ||
     (argc == 4 && atoi(argv[argc - 1]) != TRANSMITTER) ||
     ((strcmp("/dev/ttyS0", argv[1])!=0) && (strcmp("/dev/ttyS1",
argv[1])!=0))) {
    printf("Usage:\tnserial SerialPort\nex:\n\t./app /dev/ttySX fileName
1 [TRANSMITTER]\n\t./app /dev/ttySX 0 [RECEIVER]\n");
    exit(1);
```

```
  }

  /* Receive serial port and status specification */
  appLayer.status = atoi(argv[argc - 1]);
  appLayer.fileDescriptor = llopen(argv[1], appLayer.status);

  switch (appLayer.status) {
    case RECEIVER:
      res = appRead(appLayer.fileDescriptor);
      llclose(appLayer.fileDescriptor, RECEIVER);
      break;
    case TRANSMITTER:
      res = appWrite(appLayer.fileDescriptor, argv[2]);
      llclose(appLayer.fileDescriptor, TRANSMITTER);
      break;
  }

  return res;
}
```

link_layer.h:

```
#ifndef _LINK_LAYER_H
#define _LINK_LAYER_H

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include "./sender.h"
#include "./receiver.h"

/**
 * @brief Set up the serial port settings and link the transmitter and
receiver through control frames
 *
 * @param path - serial port specifier
 * @param status - transmitter or receiver flag
 * @return int - link layer identifier or -1 in case of error
 */
int llopen(char* path, int status);

/**
 * @brief Reset the serial port settings and close the PCs connection
 *
 * @param fd - link layer identifier
 * @param status - transmitter or receiver flag
```

```
 * @return int - positive number in case of success, -1 otherwise
 */
int llclose(int fd, int status);

/**
 * @brief Send a frame protected by byte stuffing with information
 *
 * @param fd - link layer identifier
 * @param buffer - character buffer to transmit
 * @param length - character buffer size
 * @return int - number of bytes sent, -1 in case of error
 */
int llwrite(int fd, unsigned char * buffer, int length);

/**
 * @brief Receive a frame protected by byte stuffing and process its
content
 *
 * @param fd - link layer identifier
 * @param buffer - read character buffer
 * @return int - number of bytes received, -1 in case of error
 */
int llread(int fd, unsigned char * buffer);

/**
 * @brief Applies the byte stuffing mechanism to the given frame buffer
 *
 * @param toStuff - frame to stuff
 * @return struct frame* - frame already stuffed
 */
struct frame * stuffing(struct frame * toStuff);

/**
 * @brief Removes the byte stuffing mechanism of the given frame buffer
 *
 * @param toDestuff - frame to destuff
 * @return struct frame* - frame already destuffed
 */
struct frame * destuffing(struct frame * toDestuff);

#endif /*_LINK_LAYER_H*/
```

link_layer.c:

```
#include "../headers/link_layer.h"

struct termios oldtio;
int flag, conta;
```

```c
struct linkLayer ll = {0, 3, 0};

int llopen(char* path, int status) {

  int fd = open(path, O_RDWR | O_NOCTTY );

  struct termios newtio;

  (void) signal(SIGALRM, sigAlarmHandler);

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

  bzero(&newtio, sizeof(newtio));
  newtio.c_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;
  newtio.c_iflag = IGNPAR;
  newtio.c_oflag = 0;

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

  newtio.c_cc[VTIME]    = 5;   /* inter-character timer unused */
  newtio.c_cc[VMIN]     = 0;   /* blocking read until 1 chars received */

  tcflush(fd, TCIOFLUSH);

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

  printf("New termios structure set\n");

  if(status) return sendControl(fd, SETUP, UAKN);

  return receiveControl(fd, SETUP, UAKN);
}

int llclose(int fd, int status) {
  int res = 0;
  switch (status) {
    case TRANSMITTER:
      if((res = sendControl(fd, DISC, DISC)) < 0) return -1;
      if((res = sendMessage(fd, A_SND, UAKN)) < 0) return -1;
      sleep(1);
      break;
    case RECEIVER:
```

```
      if((res = receiveControl(fd, DISC, DISC)) < 0) return -1;
      if((res = receiveMessage(fd, A_SND, UAKN)) < 0) return -1;
      break;
  }

  if ( tcsetattr(fd, TCSANOW, &oldtio) == -1) {
    perror("tcsetattr");
    exit(-1);
  }
  close(fd);
  return res;
}

int llwrite(int fd, unsigned char * buffer, int length) {
  unsigned char * newBuffer = (unsigned char *) malloc(length + 6);
  unsigned int newLength = length + 6;

  newBuffer[0] = FLAG;
  newBuffer[1] = A_SND;
  newBuffer[2] = SEQUENCENUMBER(ll.sequenceNumber);
  newBuffer[3] = BCC(A_SND, newBuffer[2]);

  unsigned int bcc2 = 0x0;
  for(size_t i = 0; i < length; i++) {
    newBuffer[4 + i] = buffer[i];
    bcc2 ^= buffer[i];
  }
  newBuffer[newLength - 2] = bcc2;
  newBuffer[newLength - 1] = FLAG;
  struct frame * toStuff = (struct frame *) malloc(sizeof(struct frame));
  toStuff->buffer = newBuffer;
  toStuff->length = newLength;

  struct frame * frame = stuffing(toStuff);
  free(toStuff->buffer);
  free(toStuff);

  int res = sendData(fd, frame);

  free(frame->buffer);
  free(frame);

  return res;

}

int llread(int fd, unsigned char * buffer) {
  int res;
  unsigned char buf[255];
```

```
  if((res = receiveData(fd, buf)) < 0) return -1;

  struct frame * toDestuff = (struct frame *) malloc(sizeof(struct
frame));
  toDestuff->buffer = buf;
  toDestuff->length = res;

  struct frame * frame = destuffing(toDestuff);
  free(toDestuff);

  /* Wrong header - ignore */
  if (frame->buffer[0] != FLAG || frame->buffer[1] != A_SND ||
      frame->buffer[3] != (BCC(A_SND, SEQUENCENUMBER(ll.sequenceNumber)))
) {

    free(frame->buffer);
    free(frame);
    return 0;
  }

  unsigned char prevSequenceNumber = (ll.sequenceNumber) ? 0:1;
  /* Duplicate frame - ignore */
  if(frame->buffer[2] == SEQUENCENUMBER(prevSequenceNumber)) {
    sendMessage(fd, A_RCVR, RR(ll.sequenceNumber));
    free(frame->buffer);
    free(frame);
    return 0;
  }
  /* Wrong header - ignore */
  if( frame->buffer[2] != SEQUENCENUMBER(ll.sequenceNumber)) {
    free(frame->buffer);
    free(frame);
    return 0;
  }

  unsigned char bcc2 = 0x00;
  for(size_t i = 4; i < (frame->length - 2); i++) {
    bcc2 ^= frame->buffer[i];
    buffer[i - 4] = frame->buffer[i];
  }

  /* Wrong BCC2 */
  if(frame->buffer[frame->length - 2] != bcc2) {
    sendMessage(fd, A_RCV, REJ(ll.sequenceNumber));
    free(frame->buffer);
    free(frame);
    return 0;
  }
```

```
  free(frame);

  /* Correct frame */
  ll.sequenceNumber = prevSequenceNumber;
  sendMessage(fd, A_RCVR, RR(ll.sequenceNumber));

  return res - 6;
}

struct frame * stuffing(struct frame * toStuff) {
  struct frame * newFrame = (struct frame *) malloc(sizeof(struct
frame));
  newFrame->buffer = (unsigned char *)malloc(255);

  unsigned int newIndex = 0;
  newFrame->buffer[newIndex++] = FLAG;
  for(size_t i = 1; i < toStuff->length - 1; i++) {
    if (toStuff->buffer[i] == FLAG) {
      newFrame->buffer[newIndex++] = ESC;
      newFrame->buffer[newIndex++] = (NEWVAL(FLAG));
      continue;
    }
    if(toStuff->buffer[i] == ESC) {
      newFrame->buffer[newIndex++] = ESC;
      newFrame->buffer[newIndex++] = (NEWVAL(ESC));
      continue;
    }
    newFrame->buffer[newIndex++] = toStuff->buffer[i];
  }
  newFrame->buffer[newIndex++] = FLAG;
  newFrame->length = newIndex;
  return newFrame;
}

struct frame * destuffing(struct frame * toDestuff) {
  struct frame * newFrame = (struct frame *) malloc(sizeof(struct
frame));
  newFrame->buffer = (unsigned char *)malloc(255);

  unsigned int newIndex = 1;
  newFrame->buffer[0] = FLAG;
  for(size_t i = 1; i < toDestuff->length - 1; i++) {
    if (toDestuff->buffer[i] == ESC) {
      i++;
      if(toDestuff->buffer[i] == (NEWVAL(FLAG)) ) {
        newFrame->buffer[newIndex++] = FLAG;
        continue;
      }
```

```
      newFrame->buffer[newIndex++] = ESC;
      continue;
    }
    newFrame->buffer[newIndex++] = toDestuff->buffer[i];
  }
  newFrame->buffer[newIndex++] = FLAG;
  newFrame->length = newIndex;

  return newFrame;
}
```

receiver.h:

```
#ifndef _RECEIVER_H
#define _RECEIVER_H

#include <unistd.h>
#include <signal.h>
#include "./utils.h"

/**
 * @brief Receives control frame specified in the controlField parameter
and sends a response
 *
 * @param fd - link layer identifier
 * @param controlField - control frame specifier to receive
 * @param response - control frame specifier to send
 * @return int - link layer identifier in case of success, -1 in case of
error
 */
int receiveControl(int fd, unsigned char controlField, unsigned char
response);

/**
 * @brief Simulates the state of the current frame beeing read
 *
 * @param byteReceived - received byte from serial port
 * @param currentState - current state of frame
 * @param prevWasFlag - flag indicating if previous byte read was a flag
 */
void stateMachineReceiver(char byteReceived, enum state * currentState,
int * prevWasFlag);

/**
 * @brief Reads information from serial port until end of frame
 *
 * @param fd - link layer identifier
 * @param buffer - character buffer with read information
```

```
 * @return int - buffer size
 */
int receiveData(int fd, unsigned char * buffer);

#endif /*_RECEIVER_H*/
```

receiver.c:

```
#include "../headers/receiver.h"

int receiveControl(int fd, unsigned char controlField, unsigned char
response) {

  if(receiveMessage(fd, A_SND, controlField) < 0) return -1;

  if(sendMessage(fd, A_RCVR, response) < 0) return -1;

  return fd;
}

void stateMachineReceiver(char byteReceived, enum state * currentState,
int * prevWasFlag) {
  switch (*currentState) {
    case START: {
      if(byteReceived == FLAG) {
       *currentState = FLAG_RCV;
       *prevWasFlag = 1;
       }
      break;
    }
    case FLAG_RCV: {
      if (byteReceived == FLAG) *currentState = (*prevWasFlag) ?
FLAG_RCV:STOP;
      else *prevWasFlag = 0;
      break;
    }
    default:
      break;
  }
}

int receiveData(int fd, unsigned char * buffer) {
  enum state current_state = START;
  char buf[255];
  int res = 0, current_index = 0;
  int prevWasFlag = 0;

  do {
```

```
    if((res = read(fd, buf, 1)) < 0)  {
      return -1;
    }

    if (!res) continue;

    stateMachineReceiver(buf[0], &current_state, &prevWasFlag);
    if(prevWasFlag && current_state == FLAG_RCV) current_index = 0;
    if(current_state != START) buffer[current_index++] = buf[0];
  } while(current_state != STOP);

  return current_index;
}
```

sender.h:

```
#ifndef _SENDER_H
#define _SENDER_H

#include <unistd.h>
#include <signal.h>
#include "./utils.h"

/**
 * @brief Sender alarm handler
 *
 */
void sigAlarmHandler();

/**
 * @brief Sends control frame specified in the controlField parameter
and receives a response
 *
 * @param fd - link layer identifier
 * @param controlField - control frame specifier to send
 * @param response - control frame specifier to receive
 * @return int - link layer identifier in case of success, -1 in case
of error
 */
int sendControl(int fd, unsigned char controlField, unsigned char
response);

/**
 * @brief Simulates the state of the current response frame beeing
read
 *
 * @param byteReceived - received byte from serial port
 * @param currentState - current state of frame
```

```
 * @param controlField - control field specifier which will be read
from received response
 */
void stateMachineSender(unsigned char byteReceived, enum state *
currentState, unsigned char * controlField);

/**
 * @brief Sends information to serial port and waits for receiver
response (confirmation or rejection)
 *
 * @param fd - link layer identifier
 * @param frame - frame to write to the serial port
 * @return int - number of bytes sents, -1 in case of error
 */
int sendData(int fd, struct frame * frame);

#endif /*_SENDER_H*/
```

sender.c:

```
#include "../headers/sender.h"

extern struct linkLayer ll;
int counter;

void sigAlarmHandler(){
  printf("alarme #%d\n", counter);
  ll.alarmFlag = 1;
  counter++;
}

int sendControl(int fd, unsigned char controlField, unsigned char
response) {

  if(sendMessage(fd, A_SND, controlField) < 0) return -1;

  if(receiveMessage(fd, A_RCVR, response) < 0) return -1;

  return fd;
}

void stateMachineSender(unsigned char byteReceived, enum state *
currentState, unsigned char * controlField) {
  switch (*currentState) {
      case START:
      if(byteReceived == FLAG) *currentState = FLAG_RCV;
      break;
```

```
    case FLAG_RCV:
      if(byteReceived == A_RCVR) *currentState = A_RCV;
      else if (byteReceived != FLAG) *currentState = START;
      break;
    case A_RCV:
      if(byteReceived == FLAG) *currentState = FLAG_RCV;
      else if(byteReceived == (RR(1)) || byteReceived == (RR(0)) ||
byteReceived == (REJ(0)) || byteReceived == (REJ(1))) {
        *controlField = byteReceived;
        *currentState = C_RCV;
      }
      else *currentState = START;
      break;
    case C_RCV:
      if(byteReceived == FLAG) *currentState = FLAG_RCV;
      else if(byteReceived == (BCC(A_RCVR, *controlField)))
*currentState = BCC_OK;
      else *currentState = START;
      break;
    case BCC_OK:
      if(byteReceived == FLAG) *currentState = STOP;
      else *currentState = START;
      break;
    default:
      break;
  }
}

int sendData(int fd, struct frame * frame) {
  enum state current_state = START;
  char buf[255];
  int resR = 0, resW = 0;
  ll.alarmFlag = 0;
  counter = 0;
  unsigned char controlField;

  resW = write(fd, frame->buffer, frame->length);
  /* Activates 3s alarm */
  alarm(3);

  do {
    if(ll.alarmFlag) {
      /* Activates 3s alarm */
      alarm(3);
      ll.alarmFlag = 0;
      resW = write(fd, frame->buffer, frame->length);
    }

    resR = read(fd, buf, 1);
```

```
    if (!resR) continue;
    stateMachineSender(buf[0], &current_state, &controlField);
  } while(current_state != STOP && counter <= ll.timeOutMax);

  alarm(0);

  int nextSequenceNumber = (ll.sequenceNumber) ? 0:1;
  if(current_state == STOP) {
    if( controlField == (REJ(ll.sequenceNumber)) ) {
      return sendData(fd, frame);
    }
    if( controlField == (RR(nextSequenceNumber)) ) {
      ll.sequenceNumber = nextSequenceNumber;
      return resW;
    }

  }
  return -1;
}
```

utils.h:

```
#ifndef _UTILS_H
#define _UTILS_H

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

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

#define BAUDRATE B38400
#define MODEMDEVICE "/dev/ttyS1"
#define _POSIX_SOURCE 1 /* POSIX compliant source */
#define FLAG       0x7E
#define ESC        0x7D
#define NEWVAL(x)  x ^ 0x20
#define SETUP      0x03
#define DISC       0x0B
#define UAKN       0x07
#define A_SND      0x03
#define BCC(x, y)  x ^ y
#define SEQUENCENUMBER(x) x << 6
#define A_RCVR     0x01
#define RR(x)      x<<7 | 0x05
#define REJ(x)     x<<7 | 0x01
```

```
#define TRANSMITTER 1
#define RECEIVER    0
#define DATA_PKT    0x01
#define START_PKT   0x02
#define END_PKT     0x03
#define FILE_SIZE   0
#define FILE_NAME   1

enum state {START, FLAG_RCV, A_RCV, C_RCV, BCC_OK, STOP};

struct linkLayer {
  unsigned int sequenceNumber;
  unsigned int timeOutMax;
  unsigned int alarmFlag;
};

struct frame
{
  unsigned char * buffer;
  int length;
};

/**
 * @brief Simulates the state of the current response frame beeing read
 *
 * @param byteReceived - received byte from serial port
 * @param currentState - current state of frame
 * @param addressField - address field specifier which is expected to be
read from received response
 * @param controlField - control field specifier which is expected to be
read from received response
 */
void stateMachineSU(char byteReceived, enum state * currentState,
unsigned char addressField, unsigned char controlField);

/**
 * @brief Sends control frame with given parameters
 *
 * @param fd - link layer identifier
 * @param adressField - address field specifier
 * @param controlField - control field specifier
 * @return int - bytes send, -1 in case of error
 */
int sendMessage(int fd, unsigned char adressField, unsigned char
controlField);

/**
 * @brief Receives control frame with given parameters
 *
```

```
 * @param fd - link layer identifier
 * @param adressField - address field specifier
 * @param controlField - control field specifier
 * @return int - positive in case of success, -1 otherwise
 */
int receiveMessage(int fd, unsigned char adressField, unsigned char
controlField);

#endif /*_UTILS_H*/
```

utils.c:

```
#include "../headers/utils.h"

void stateMachineSU(char byteReceived, enum state * currentState,
unsigned char addressField, unsigned char controlField) {
  switch (*currentState) {
    case START:
      if(byteReceived == FLAG) *currentState = FLAG_RCV;
      break;
    case FLAG_RCV:
      if(byteReceived == addressField) *currentState = A_RCV;
      else if (byteReceived != FLAG) *currentState = START;
      break;
    case A_RCV:
      if(byteReceived == FLAG) *currentState = FLAG_RCV;
      else if(byteReceived == controlField) *currentState = C_RCV;
      else *currentState = START;
      break;
    case C_RCV:
      if(byteReceived == FLAG) *currentState = FLAG_RCV;
      else if(byteReceived == (BCC(addressField, controlField)))
*currentState = BCC_OK;
      else *currentState = START;
      break;
    case BCC_OK:
      if(byteReceived == FLAG) *currentState = STOP;
      else *currentState = START;
      break;
    default:
      break;
  }
}

int sendMessage(int fd, unsigned char adressField, unsigned char
controlField) {
  unsigned char message[5];

  message[0] = FLAG;
```

```
  message[1] = adressField;
  message[2] = controlField;
  message[3] = BCC(adressField, controlField);
  message[4] = FLAG;

  return write(fd, message, 5);
}

int receiveMessage(int fd, unsigned char adressField, unsigned char
controlField) {
  enum state current_state = START;
  char buf[255];
  int res = 0;
  do {
    if((res = read(fd, buf, 1)) < 0) return -1;
    if (!res) continue;
    stateMachineSU(buf[0], &current_state, adressField, controlField);
  } while(current_state != STOP);

  return res;
}
```

ANEXO II

Nº total de bytes	10968
Nº total de bits	87744
C (Baudrate)	38400
Tamanho de pacote (bytes)	100

***Tabela** 1: Valores constantes*

Tamanho do pacote (Bytes)	Tempo (s)	S Medido
50	4,764	0,479638959
100	4,325	0,528323699
150	4,182	0,546389287
200	4,106	0,556502679

***Tabela 2**: Cálculo da eficiência em função do tamanho dos pacotes*

FER(%)	Tf (ms) Médio	Tprop (ms)	Tempo (s)	S Teórica	S Medido
5%	29,665	10	23,014	0,567436827	0,09928739
10%	29,671	10	33,583	0,53761551	0,06804038
15%	29,656	10	54,62	0,507644595	0,04183449
20%	29,661	10	79,239	0,47781559	0,02883681
25%	29,661	10	110,951	0,447952115	0,02059468

***Tabela 3**: Cálculo da eficiência em função do erro nas tramas*

Tprop (ms)	Tf (ms) Médio	a	S Teórico	S Medido	Tempo (s)
5	29,703	0,168333165	0,748129864	0,466612212	4,897
10	29,68	0,336927224	0,59742351	0,418958563	5,454
15	29,667	0,505612296	0,497209513	0,379946791	6,014
20	29,672	0,674036128	0,425881272	0,347581381	6,574
25	29,676	0,842431595	0,372458457	0,320297168	7,134

***Tabela 4**: Cálculo da eficiência em função do a (variação de Tprop)*