Redes de Computadores

Protocolo de Ligação de Dados

Mestrado Integrado em Engenharia Informática e Computação

Gonçalo Teixeira up201806562@fe.up.pt
Gonçalo Alves up201806451@fe.up.pt

24 de outubro de 2020

Índice

Sumário

Este relatório foi elaborado no âmbito da unidade curricular de Redes e Computadores, e trata-se da implementação de um protocolo de transferência de dados. O trabalho consiste no desenvolvimento de uma aplicação capaz de transferir ficheiros de um computador para o outro através de uma porta série.

O trabalho foi concluído e a aplicação desenvolvida é capaz de transferir ficheiros sem perda de dados.

Introdução

O objetivo deste trabalho consistiu na implementação de um protocolo de ligação de dados, de acordo com o guião fornecido, e no teste do dito protocolo, através de uma aplicação de transferência de dados. Quanto ao relatório, o seu objetivo é detalhar a componente teórica do trabalho, com a estrutura descrita em baixo:

Arquitetura - Descrição dos blocos funcionais e das interfaces implementadas
Estrutura do código - Descrição das APIs, principais estruturas de dados e funções e as suas relações com a arquitetura
Casos de uso principais - Identificação dos casos de uso; Sequências de chamada de funções
Protocolo de ligação lógica - Identificação dos principais aspetos funcionais; Descrição da estratégia de implementação destes, com a exibição de extratos de código
Protocolo de aplicação - Identificação dos principais aspetos funcionais; Descrição da estratégia de implementação destes, com a exibição de extratos de código
Validação - Descrição dos testes efetuados; Apresentação quantificada dos resultados
Eficiência do protocolo de ligação de dados - Caraterização estatística da eficiência do protocolo, feita com recurso a medidas sobre o código desenvolvido
Conclusão - Síntese da informação apresentada nas secções anteriores; Reflexão sobre os objetivos de aprendizagem alcançados.

Arquitetura

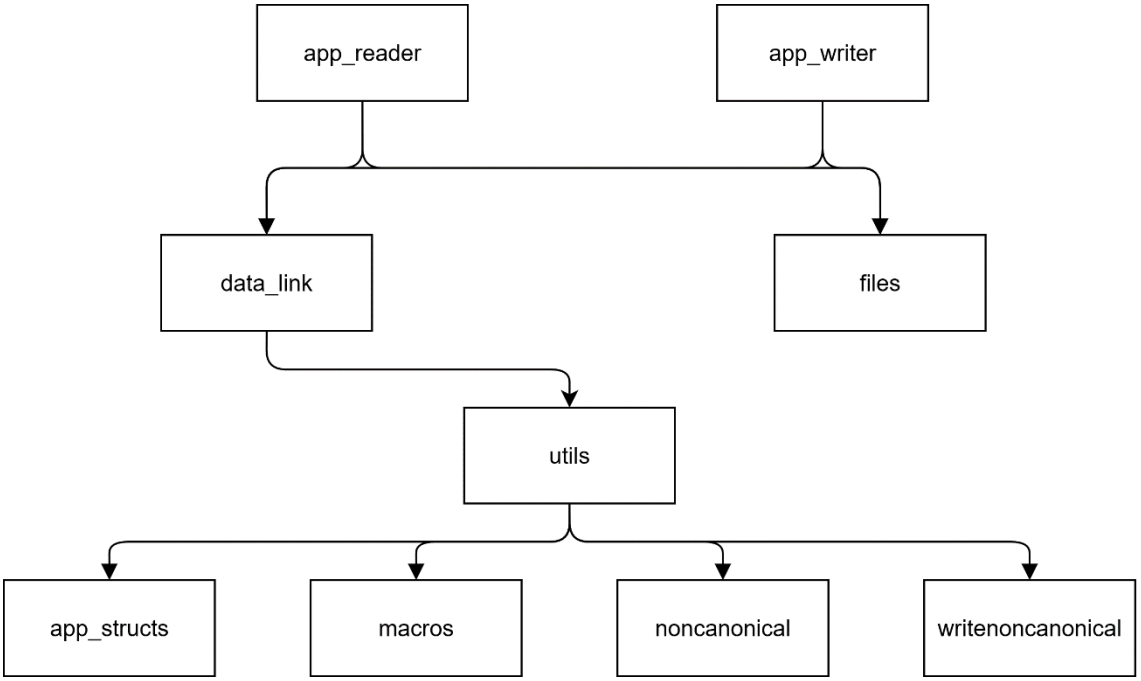
O trabalho está dividido em duas camadas: a camada do de ligação de dados e a camada da aplicação.

A camada de ligação de dados é responsável pelas interações com a porta de série, tal como: a abertura, o fecho, a leitura e a escrita desta. Além disso, a camada também é responsável pelo tratamento das tramas: delineação, *stuffing*, proteção e retransmissão destas.

A camada da aplicação é responsável pelo envio e receção de ficheiros, fazendo uso da interface da camada de ligação de dados. Além disso, a camada também é responsável pelo processamento do serviço: tratamento de cabeçalhos, distinção entre pacotes de controlo e de dados e numeração destes.

Estrutura de código

O código está dividido em diversos ficheiros de código, tendo uma hierarquia associada a estes, como se demonstra no diagrama seguinte:



Assim, a estrutura do código será feita de forma hierárquica, de baixo para cima.

Writenoncanonical

- atende – *Handler* do sinal SIGALRM;
- open_writer – Abre a porta série do emissor e implementa o *handler* do alarme;
- close_writer – Fecha a porta série do emissor;

Noncanonical

- open_reader – Abre a porta série do recetor;
- close_reader – Fecha a porta série do recetor;

Macros

O ficheiro macros.h contém, tal como o nome indica, macros importantes para o programa, entre as quais:

- TRIES – número de tentativas de escrita;
- TIMEOUT – número de segundos de espera por resposta;

App_structs

O ficheiro app_structs.h contém as estruturas de dados relevantes para o programa:

- information_frame_t – *Frame* contendo: *address*, *control*, *bcc*, *data*, *data_size*, *bcc2* e *raw_bytes*;
- data_packet_t – Pacote de dados contendo: *control*, *sequence*, *data_field_size*, *data*, *raw_bytes* e *raw_bytes_size*;
- control_packet_t – Pacote de dados contendo: *control*, *file_size*, *file_name*, *filesize_size*, *raw_bytes* e *raw_bytes_size*;
- file_t – Ficheiro contendo: *data*, *name* e *size*;

Utils

- parse_control_packet – Função que faz o *parse* de bytes para a estrutura control_packet_t;
- parse_data_packet – Função que faz o *parse* de bytes para a estrutura data_packet_t;
- print_control_packet – Função que imprime um pacote de controlo;
- print_data_packet – Função que imprime um pacote de dados. Além disso, permite mostrar todos os bytes de dados, através do uso de uma flag;
- print_elapsed_time – Função que imprime o tempo passado entre uma escrita e uma leitura;
- verify_message – Função que verifica erros numa trama de Informação, através do bcc;
- print_message – Função que imprime uma trama de Informação;
- check_connection– Função que verifica se a porta série não fechou;
- array_to_number– Função que transforma um *array* num número de 8 bytes;
- number_to_array– Função que transforma um número de 8 bytes num *array*;

Data_link

- send_supervision_frame – Recebe uma porta e envia-lhe uma mensagem de controlo;
- receive_supervision_frame – Recebe uma porta e lê uma mensagem de controlo dela;
- receive_acknowledgment – Recebe uma mensagem de ACK e retorna o seu *byte* de controlo;
- send_acknowledgment – Envia uma mensagem de ACK;
- receive_set – Função que espera receber uma mensagem SET e envia UA de seguida;
- send_set – Função que envia uma mensagem SET e espera receber UA de seguida;
- disconnect_writer – Função que envia uma mensagem de DISC, espera receber DISC de volta e envia UA;
- disconnect_reader - Função que recebe uma mensagem de DISC, espera enviar DISC de volta e recebe UA;
- llopen – Função que abre a porta série do emissor/recetor e inicia a ligação de dados, retornando o descritor correspondente;
- llclose – Função que fecha a porta série do emissor/recetor e termina a ligação de dados;
- llwrite – Função que faz o *stuffing* de um pacote de dados e envia para o recetor, esperando receber uma mensagem de ACK de volta e procede de acordo com esta;
- llread – Função que lê uma mensagem do emissor, faz o *destuffing*, verifica a mensagem e envia uma mensagem ACK adequada;

Files

- get_file_size – Função que calcula o tamanho de um ficheiro, em bytes;
- read_file – Função que lê os dados de um ficheiro;
- split_file – Função que obtém bytes de um ficheiro entre um índice inicial e final, inclusive;
- join_file – Função que concatena pacotes;
- write_file – Função que cria uma cópia do ficheiro recebido;

App_writer

- generate_control_packet – Função que cria um pacote de controlo;
- generate_data_packet – Função que cria um pacote de dados;
- main – Função responsável pela escrita de um ficheiro;

App_reader

- main – Função responsável pela leitura de um ficheiro;

Casos de uso principais

Os casos de uso principais da aplicação são: a interface, que permite a escolha do ficheiro que o emissor pretende enviar, e a transferência do ficheiro, através da porta série.

De modo a dar-se a transferência do ficheiro, o utilizador necessitará de introduzir o número da porta série a ser utilizada, como por exemplo 11. Adicionalmente, caso se trate do emissor, também terá de inserir o ficheiro a ser enviado, como por exemplo pinguim.gif

A transmissão de dados dá-se pela seguinte ordem:

- Abertura da ligação entre os computadores;
- Geração dos pacotes START e STOP, para controlo;
- Escrita do pacote START;
- Escrita dos pacotes de dados;
- Escrita do pacote STOP;
- Fecho da ligação entre os computadores;

A receção de dados dá-se pela seguinte ordem:

- Abertura da ligação entre os computadores;
- Leitura e impressão do pacote START;
- Leitura e impressão dos pacotes de dados;
- Leitura e impressão do pacote STOP;
- Impressão da mensagem completa;
- Fecho da ligação entre os computadores;

Protocolo de ligação lógica

llopen

Esta função tem a responsabilidade de estabelecer a ligação entre o emissor e o recetor.

No caso do emissor, a porta série é aberta, é enviada uma mensagem SET e é esperada uma mensagem UA.

No caso do recetor, a porta de série é aberta, é recebida uma mensagem SET e é enviada uma mensagem UA.

llclose

Esta função tem a responsabilidade de fechar a ligação entre o emissor e o recetor.

No caso do emissor, é enviada uma mensagem DISC, posteriormente é recebida uma mensagem igual de volta, é enviada uma mensagem UA e a ligação termina.

No caso do recetor, é recebida uma mensagem DISC, de seguida é enviada essa mesma mensagem para o recetor, é esperada uma mensagem UA e a ligação é terminada.

llwrite

Esta função é responsável pelo envio de tramas.

Inicialmente, é composto o *header* da mensagem: *address*, *control* e *bcc1*. De seguida, é feito o *stuffing* da mensagem e a construção do *bcc2* (também com

stuffing). Finalmente, a função irá enviar a mensagem completa para o emissor e esperar pela mensagem ACK. Dependendo desta, o emissor poderá: continuar a transmissão do ficheiro, passando para o próximo pacote; retransmitir o pacote acabado de enviar, devido a um erro. A retransmissão de um pacote também se pode dar quando o tempo de espera de uma resposta exceder o tempo máximo de espera, TIMEOUT.

llread
Esta é função é responsável pela receção de tramas.

Inicialmente, é feita uma leitura da porta série, caractere a caractere. De seguida, é feito o *destuffing* da mensagem e esta é guardada numa estrutura de dados (information_frame_t). Finalmente, é feita uma verificação de erros e, dependendo do resultado desta, é enviada a mensagem adequada para o emissor.

Protocolo de aplicação
O protoloco de aplicação implementado tem como aspetos principais:

- Envio de pacotes de controlo START e STOP. Estes contêm o nome e o tamanho do ficheiro a ser enviado;
- Divisão do ficheiro em pacotes, no emissor, e a concatenação dos pacotes recebidos, no recetor;
- Encapsulamento de cada pacote de dados com um *header* contendo o número de sequência do pacote e o tamanho do pacote;
- Leitura do ficheiro a enviar, no emissor, e criação do ficheiro, no recetor.

Estas funcionalidades foram implementadas usando funções descritas a seguir.

generate_control_packet
Esta função retorna um pacote START ou STOP, recebendo como argumento um inteiro de modo a identificar o tipo de pacote. Estes pacotes serão enviados usando a função llwrite, pertencente ao protocolo de ligação de dados.

generate_data_packet
Esta função retorna um pacote de dados, recebendo como argumentos: dados de um ficheiro, tamanho dos dados e o número de sequência. Estes pacotes serão enviados usando a função llwrite, pertencente ao protocolo de ligação de dados.

split_file
Esta função retorna dados, recebendo como argumentos: o ficheiro de onde se quer obter os dados, o índice do primeiro byte a recolher e o índice do último byte a receber. Estes dados serão usados na função acima mencionada.

get_file_size
Esta função retorna o tamanho de um ficheiro, recebendo como argumento o descritor de um ficheiro.

read_file
Esta função retorna os dados de um ficheiro, recebendo como como argumentos: o descritor de um ficheiro e o tamanho deste.

join_file
Esta função recebe como argumentos: um *array* onde se vão concatenar os pacotes, um pacote, o tamanho do pacote e o índice dos dados.

write_file
Esta função cria uma cópia do ficheiro recebido, recebendo com argumentos: o nome do ficheiro, os *bytes* deste e o seu tamanho.

Validação
Ainda não foram feitos testes

Eficiência do protocolo de ligação de dados
Variação do FER
Variação do tamanho das tramas I
Variação da capacidade da ligação (C)

Conclusões
O tema deste trabalho foi a criação de um protocolo de ligação de dados, que consiste em fornecer um serviço de comunicação de dados fiável entre dois sistemas ligados por um meio de transmissão, neste caso, um cabo série.

Adicionalmente, foi nos dado a conhecer a independência entre camadas, e cada um dos blocos funcionais da arquitetura da aplicação desenvolvida cumpre esta independência.

Na camada de ligação de dados não é feito qualquer processamento que incida sobre o cabeçalho dos pacotes a transportar em tramas de Informação. Ao nível da ligação de dados não existe qualquer distinção entre pacotes de controlo e de dados, nem é relevante (nem tida em conta) a numeração dos pacotes de dados.

Na camada de aplicação não são conhecidos os detalhes do protocolo de ligação de dados, mas apenas a forma como se acede ao serviço. O protocolo de aplicação desconhece a estrutura das tramas e o respetivo mecanismo de delineação, a existência de *stuffing* (e qual a opção adotada), o mecanismo de proteção das tramas, eventuais retransmissões de tramas de Informação, etc.

Concluindo, o protocolo de ligação de dados foi realizado com sucesso, tendo-se cumprido todos os objetivos, e o desenvolvimento deste contribuiu para um aprofundamento do conhecimento, tanto teórico como prático, deste tema.

Anexo I – Código fonte

app_reader.c

```c
#include "data_link.h"
#include "files.h"

file_t file;

int main(int argc, char *argv[]) {
  if (argc != 2) {
    printf("Usage: %s <number for serial port>\n", argv[0]);
    printf("\nExample: %s 11\t-\tfor '/dev/ttyS11'\n", argv[0]);
    exit(ERROR);
  }

  /* opens transmiter file descriptor on second layer */
  int receiver_fd = llopen(atoi(argv[1]), RECEIVER);
  /* in case there's an error oppening the port */
  if (receiver_fd == ERROR) {
    exit(ERROR);
  }

  char buffer[1024];
  int size;

  int state = 0;

  // * START Control Packet
  while (state == 0) {
    memset(buffer, 0, sizeof(buffer));
    while ((size = llread(receiver_fd, buffer)) == ERROR) {
      printf("Error reading\n");
    }
    control_packet_t packet = parse_control_packet(buffer, size);

    file.size = array_to_number(packet.file_size, packet.filesize_size);
    file.name = packet.file_name;

    print_control_packet(packet);
    if (packet.control == START) {
      state = 1;
    }
  }

  // * DATA Packets
  unsigned char *full_message = (unsigned char*) malloc (file.size);
  int index = 0;

  while (state == 1) {
    memset(buffer, 0, sizeof(buffer));
    while ((size = llread(receiver_fd, buffer)) == ERROR) {
      printf("Error reading\n");
    }
    if (buffer[0] == STOP) {
      state = 2;
      break;
    }
    data_packet_t data = parse_data_packet(buffer, size);
```

```
    if (data.control != DATA) continue;

    print_data_packet(&data, FALSE);
    join_file(full_message, data.data, data.data_field_size, index);
    index += data.data_field_size;
  }

  // * STOP Control Packet
  if (state == 2) {
    control_packet_t packet = parse_control_packet(buffer, size);
    print_control_packet(packet);

    write_file("pinguim_clone.gif", full_message, file.size);
    printf("Received file\n");
  }

  /* resets and closes the receiver fd for the port */
  llclose(receiver_fd, RECEIVER);

  return 0;
}
```

app_structs.h

```
typedef struct {
  unsigned char address;
  unsigned char control;
  unsigned char bcc1;
  unsigned char *data;
  int data_size; /* size of the data array */
  unsigned char bcc2;

  unsigned char *raw_bytes; /* full set of bytes for the message */
} information_frame_t;

typedef struct {
  unsigned char control;
  unsigned char sequence;
  int data_field_size;
  unsigned char data[1024];

  unsigned char *raw_bytes;
  int raw_bytes_size;
} data_packet_t;

typedef struct {
  unsigned char control;
  unsigned char *file_size;
  unsigned char *file_name;
  unsigned int filesize_size;

  unsigned char *raw_bytes;
  int raw_bytes_size;
} control_packet_t;

typedef struct {
  unsigned char* data;
  unsigned char* name;
  unsigned long size;
```

```
} file_t;
```

app_writer.c

```
#include "data_link.h"
#include "files.h"

extern int flag;
FILE *fp;
file_t file;

control_packet_t generate_control_packet(int control) {
  control_packet_t c_packet;
  c_packet.control = control;
  c_packet.file_name = file.name;

  unsigned char buf[sizeof(unsigned long)];
  int num = number_to_array(file.size, buf);

  c_packet.file_size = (unsigned char *)malloc(num);
  memcpy(c_packet.file_size, buf, num);
  c_packet.filesize_size = num;

  int i = 0;
  // control packet
  c_packet.raw_bytes = (unsigned char *)malloc(i + 1);
  c_packet.raw_bytes[i++] = c_packet.control;
  c_packet.raw_bytes = (unsigned char *)realloc(c_packet.raw_bytes, (i +
1));
  // file size
  c_packet.raw_bytes[i++] = FILE_SIZE;
  c_packet.raw_bytes = (unsigned char *)realloc(c_packet.raw_bytes, (i +
1));
  c_packet.raw_bytes[i++] = c_packet.filesize_size;

  for (int j = 0; j < c_packet.filesize_size; j++) {
    c_packet.raw_bytes = (unsigned char *)realloc(c_packet.raw_bytes, (i
+ 1));
    c_packet.raw_bytes[i++] = c_packet.file_size[j];
  }
  c_packet.raw_bytes = (unsigned char *)realloc(c_packet.raw_bytes, (i +
1));
  // file name
  c_packet.raw_bytes[i++] = FILE_NAME;
  c_packet.raw_bytes = (unsigned char *)realloc(c_packet.raw_bytes, (i +
1));
  c_packet.raw_bytes[i++] = strlen(c_packet.file_name);
  c_packet.raw_bytes = (unsigned char *)realloc(c_packet.raw_bytes, (i +
1));
  for (int j = 0; j < strlen(c_packet.file_name); j++) {
    c_packet.raw_bytes[i++] = c_packet.file_name[j];
    c_packet.raw_bytes = (unsigned char *)realloc(c_packet.raw_bytes, (i
+ 1));
  }

  c_packet.raw_bytes_size = i;

  return c_packet;
}
```

```
data_packet_t generate_data_packet(unsigned char *buffer, int size, int
sequence) {
  data_packet_t d_packet;
  d_packet.control = DATA;
  d_packet.data_field_size = size;
  d_packet.sequence = sequence;

  int i = 0;
  // control
  d_packet.raw_bytes = (unsigned char *)malloc(i + 1);
  d_packet.raw_bytes[i++] = d_packet.control;
  d_packet.raw_bytes = (unsigned char *)realloc(d_packet.raw_bytes, (i +
1));
  // sequence
  d_packet.raw_bytes[i++] = d_packet.sequence;
  d_packet.raw_bytes = (unsigned char *)realloc(d_packet.raw_bytes, (i +
1));
  // size
  unsigned int x = (unsigned int)size;
  unsigned char high = (unsigned char)(x >> 8);
  unsigned char low = x & 0xff;
  d_packet.raw_bytes[i++] = high;
  d_packet.raw_bytes = (unsigned char *)realloc(d_packet.raw_bytes, (i +
1));
  d_packet.raw_bytes[i++] = low;
  d_packet.raw_bytes = (unsigned char *)realloc(d_packet.raw_bytes, (i +
1));
  // data
  for (int j = 0; j < size; j++) {
    d_packet.data[j] = buffer[j];
    d_packet.raw_bytes[i++] = buffer[j];
    d_packet.raw_bytes = (unsigned char *)realloc(d_packet.raw_bytes, (i
+ 1));
  }

  d_packet.raw_bytes_size = i;

  return d_packet;
}

int main(int argc, char *argv[]) {
  if (argc != 2) {
    printf("Usage: %s <number for serial port>\n", argv[0]);
    printf("\nExample: %s 11\t-\tfor '/dev/ttyS11'\n", argv[0]);
    return -1;
  }

  /* opens transmiter file descriptor on second layer */
  int transmiter_fd = llopen(atoi(argv[1]), TRANSMITTER);
  /* in case there's an error oppening the port */
  if (transmiter_fd == ERROR) {
    exit(ERROR);
  }

  fp = fopen("pinguim.gif", "rb");
  file.name = "pinguim.gif";
  file.size = get_file_size(fp);
  file.data = read_file(fp, file.size);

  control_packet_t c_packet_start = generate_control_packet(START);
```

```
  control_packet_t c_packet_stop = generate_control_packet(STOP);

  // sending control packet
  struct timespec start;
  clock_gettime(CLOCK_MONOTONIC_RAW, &start);

  print_control_packet(c_packet_start);

  int size = llwrite(transmiter_fd, c_packet_start.raw_bytes,
c_packet_start.raw_bytes_size);
  if (size == ERROR) {
    printf("Error writing START Control Packet, aborting...\n");
    llclose(transmiter_fd, TRANSMITTER);
    return ERROR;
  }
  print_elapsed_time(start);

  unsigned long bytes_left = file.size;
  int index_start;
  int index_end = -1;
  int sequence = 0;
  while (bytes_left != 0 && index_end != file.size - 1) {
    usleep(STOP_AND_WAIT);

    index_start = index_end + 1;
    if (bytes_left >= 1023) {
      index_end = index_start + 1023;
    } else {
      index_end = index_start + bytes_left - 1;
    }
    bytes_left -= (index_end - index_start) + 1;

    clock_gettime(CLOCK_MONOTONIC_RAW, &start);

    data_packet_t data = generate_data_packet(split_file(file.data,
index_start, index_end), index_end - index_start + 1, sequence++);
    print_data_packet(&data, FALSE);

    size = llwrite(transmiter_fd, data.raw_bytes, data.raw_bytes_size);
    if (size == ERROR) {
      printf("Error writing Data Packet, aborting...\n");
      llclose(transmiter_fd, TRANSMITTER);
      return ERROR;
    }
    print_elapsed_time(start);
  }

  usleep(STOP_AND_WAIT);
  print_control_packet(c_packet_stop);
  clock_gettime(CLOCK_MONOTONIC_RAW, &start);
  size = llwrite(transmiter_fd, c_packet_stop.raw_bytes,
c_packet_stop.raw_bytes_size);
  if (size == ERROR) {
    printf("Error writing STOP Control Packet, aborting...\n");
    llclose(transmiter_fd, TRANSMITTER);
    return ERROR;
  }

  usleep(STOP_AND_WAIT);
  print_elapsed_time(start);
```

```
  /* resets and closes the receiver fd for the port */
  llclose(transmiter_fd, TRANSMITTER);

  return OK;
}
```

data_link.c

```
#include "data_link.h"

int reetransmit = 1;
extern int flag;
extern int conta;

int send_supervision_frame(int fd, unsigned char msg, unsigned char
address) {
  unsigned char mesh[5];
  mesh[0] = FLAG;
  mesh[1] = address;
  mesh[2] = msg;
  mesh[3] = mesh[1] ^ mesh[2];
  mesh[4] = FLAG;
  int err = write(fd, mesh, 5);
  if (!(err == 5))
    return ERROR;
  return 0;
}

unsigned char receive_acknowledgement(int fd) {
  // ! Remove comments if you want to debug the data being read
  int part = 0;
  unsigned char rcv_msg;
  unsigned char ctrl;
  unsigned char address;
  printf("Reading ACK supervision frame...\n");
  while (part != 5) {
    int rd = read(fd, &rcv_msg, 1);
    if (rd == -1 && errno == EINTR) {
      printf("READ failed\n");
      return 2;
    }
    switch (part) {
    case 0:
      if (rcv_msg == FLAG) {
        part = 1;
        // printf("FLAG: 0x%x\n",rcv_msg);
      }
      break;
    case 1:
      if (rcv_msg == A_1 || rcv_msg == A_3) {
        address = rcv_msg;
        part = 2;
        // printf("A: 0x%x\n",rcv_msg);
      } else {
        if (rcv_msg == FLAG)
          part = 1;
        else
          part = 0;
      }
      break;
```

```
    case 2:
      if ((rcv_msg == C_RR0) || (rcv_msg == C_RR1) || (rcv_msg == C_REJ0)
||
          (rcv_msg == C_REJ1)) {
        part = 3;
        // printf("Control: 0x%x\n",rcv_msg);
        ctrl = rcv_msg;
      } else
        part = 0;
      break;
    case 3:
      if (rcv_msg == (address ^ ctrl)) {
        part = 4;
        // printf("Control BCC: 0x%x\n",rcv_msg);
      } else
        part = 0;
      break;
    case 4:
      if (rcv_msg == FLAG) {
        part = 5;
        // printf("FINAL FLAG: 0x%x\nReceived Control\n",rcv_msg);
      } else
        part = 0;
      break;
    default:
      break;
    }
  }
  return ctrl;
}

int receive_supervision_frame(int fd, unsigned char msg) {
  // ! Remove comments if you want to debug the data being read
  int part = 0;
  unsigned char rcv_msg;
  unsigned char address;

  printf("Reading supervision frame...\n");
  while (part != 5) {
    int rd = read(fd, &rcv_msg,1);
    if (rd == -1 && errno == EINTR) {
      printf("READ failed\n");
      return 2;
    }
    switch (part) {
    case 0:
      if (rcv_msg == FLAG) {
        part = 1;
        // printf("FLAG: 0x%x\n",rcv_msg);
      }
      break;
    case 1:
      if (rcv_msg == A_1 || rcv_msg == A_3) {
        address = rcv_msg;
        part = 2;
        // printf("A: 0x%x\n",rcv_msg);
      } else {
        if (rcv_msg == FLAG)
          part = 1;
        else
```

```
          part = 0;
      }
      break;
    case 2:
      if (rcv_msg == msg) {
        part = 3;
        // printf("Control: 0x%x\n",rcv_msg);
      } else
        part = 0;
      break;
    case 3:
      if (rcv_msg == (address ^ msg)) {
        part = 4;
        // printf("Control BCC: 0x%x\n",rcv_msg);
      } else
        part = 0;
      break;
    case 4:
      if (rcv_msg == FLAG) {
        part = 5;
        // printf("FINAL FLAG: 0x%x\nReceived Control\n",rcv_msg);
      } else
        part = 0;
      break;
    default:
      break;
    }
  }
  return (part == 5) ? 0 : -1;
}

int receive_set(int fd) {
  if (receive_supervision_frame(fd, SET) == 0) {
    printf("Sending UA reply...\n");
    send_supervision_frame(fd, UA, A_3);
  }
  return 0;
}

int send_set(int fd) {
  struct timespec start;
  do {
    clock_gettime(CLOCK_MONOTONIC_RAW, &start);
    if (send_supervision_frame(fd, SET, A_3) == -1) {
      printf("Error writing SET\n");
      continue;
    }
    alarm(TIMEOUT); // activa alarme de 3s
    printf("Sent SET frame\n");
    flag = 0;
    printf("Receiving UA response...\n");
    while (!flag) {
      if (receive_supervision_frame(fd, UA) == 0) {
        reetransmit = 0;
        break;
      }
    }

    if (flag)
      printf("Timed Out - Retrying\n");
```

```
    print_elapsed_time(start);

  } while (conta < 4 && flag);

  alarm(RESET_ALARM);

  if (conta == 4) {
    reetransmit = 2;
    printf("Gave up\n");
    return -1;
  }

  return 0;
}

int disconnect_writer(int fd) {
  struct timespec start;
  do {
    clock_gettime(CLOCK_MONOTONIC_RAW, &start);
    if (send_supervision_frame(fd, DISC, A_3) == -1) {
      printf("Error writing DISC\n");
      continue;
    }
    alarm(TIMEOUT); // activa alarme de 3s
    printf("Sent DISC frame\n");
    flag = 0;
    printf("Receiving DISC response...\n");
    while (!flag) {
      if (receive_supervision_frame(fd, DISC) == 0) {
        reetransmit = 0;
        break;
      }
    }

    if (flag)
      printf("Timed Out - Retrying\n");
    print_elapsed_time(start);

  } while (conta < 4 && flag);

  alarm(RESET_ALARM);

  if (conta == 4) {
    reetransmit = 2;
    printf("Gave up\n");
    return -1;
  }

  printf("Sending UA ACK...\n");
  return send_supervision_frame(fd, UA, A_1);
}

int disconnect_reader(int fd) {
  struct timespec start;
  do {
    clock_gettime(CLOCK_MONOTONIC_RAW, &start);
    alarm(TIMEOUT); // activa alarme de 3s
    flag = 0;
    printf("Receiving DISC from writer...\n");
    while (!flag) {
```

```
      if (receive_supervision_frame(fd, DISC) == 0) {
        reetransmit = 0;
        break;
      }
    }
    printf("DISC received, sending DISC..\n");
    if (send_supervision_frame(fd, DISC, A_3) == -1) {
      printf("Error writing DISC\n");
      continue;
    }

    if (flag)
      printf("Timed Out - Retrying\n");
    print_elapsed_time(start);

  } while (conta < 4 && flag);

  printf("Receiving UA...\n");

  return receive_supervision_frame(fd, UA);
}

int send_acknowledgement(int fd, int frame, int accept) {
  printf("Sending acknowledgement...\n");
  if (frame == 0) {
    if (accept == 1) {
      // caso seja o frame 0 e seja aceite então pede o frame 1
      send_supervision_frame(fd, C_RR1, A_3);
    } else {
      send_supervision_frame(fd, C_REJ0, A_3);
    }
  } else {
    if (accept == 1) {
      send_supervision_frame(fd, C_RR0, A_3);
    } else {
      send_supervision_frame(fd, C_REJ1, A_3);
    }
  }
  return 0;
}

int llopen(int port, int type) {
  char file[48];
  sprintf(file, "/dev/ttyS%d", port);

  int fd;
  if (type == TRANSMITTER) {
    if ((fd = open_writer(file)) == ERROR) {
      perror("llopen: error on open_writer");
      return ERROR;
    }
    if (send_set(fd) == ERROR) {
      perror("llopen: error sending SET");
      return ERROR;
    }
    return fd;
  }
  else if (type == RECEIVER) {
    if ((fd = open_reader(file)) == ERROR) {
      perror("llopen: error on open_reader");
```

```
      return ERROR;
    }
    if (receive_set(fd) == ERROR) {
      perror("llopen: error receiving SET");
      return ERROR;
    }
    return fd;
  }
  perror("llopen: type not valid");
  return ERROR;
}

int llclose(int fd, int type) {
  printf("\nDisconnecting...\n");
  if (type == TRANSMITTER) {
    if (disconnect_writer(fd) == ERROR) {
      perror("llclose: error disconnecting writer: ");
      return ERROR;
    }
    if (close_writer(fd) != ERROR) {
      printf("Writer Successfully Closed!\n");
      return OK;
    } else {
      perror("llclose: writer not closed successfully: ");
      return ERROR;
    }
  }

  else if (type == RECEIVER) {
    if (disconnect_reader(fd) == ERROR) {
      perror("llclose: error disconnecting reader: ");
      return ERROR;
    }
    if (close_reader(fd) != ERROR) {
      printf("Reader Successfully Closed!\n");
      return OK;
    } else {
      perror("llclose: reader not closed successfully: ");
      return ERROR;
    }
  }
  return ERROR;
}

int llwrite(int fd, char *buffer, int length) {
  printf("Sending data...\n");
  // printf("Message: %s\n", buffer);
  printf("Coding message...\n");

  information_frame_t frame; // to keep everything organized

  frame.address = A_3;

  /* C byte - Controls package, alternating between 0 and 1*/
  frame.control = (current_frame == 0) ? C_I0 : C_I1;

  frame.bcc1 = frame.address ^ frame.control;

  int size_info = length;
```

```
  unsigned char *information_frame = (unsigned char *) malloc (size_info
*sizeof(unsigned char));
  unsigned char bcc = 0xff;
  int i = 0;
  for (int j = 0; j < length; j++) {
    /* Data stuffing and buffer size adjusting*/
    if (buffer[j] == ESCAPE) {
      information_frame =
          (unsigned char *)realloc(information_frame, ++size_info);
      information_frame[i++] = ESCAPE;
      information_frame[i++] = ESCAPE_ESC;
    } else if (buffer[j] == FLAG) {
      information_frame =
          (unsigned char *)realloc(information_frame, ++size_info);
      information_frame[i++] = ESCAPE;
      information_frame[i++] = ESCAPE_FLAG;
    } else
      information_frame[i++] = buffer[j];

    bcc = buffer[j] ^ bcc;
  }
  frame.data = information_frame; /* saves the stuffed data-buffer on the
struct */
  frame.data_size = i;            /* size of the suffed data-buffer */
  frame.bcc2 = bcc;               /* this BCC2 is not stuffed yet and it
will be displayed *unstuffed* */

  /* Saving all data to be transmitted to .raw_bytes */
  frame.raw_bytes = (unsigned char *)malloc((frame.data_size + 10) *
sizeof(unsigned char *));
  int j = 0;
  frame.raw_bytes[j++] = FLAG;
  frame.raw_bytes[j++] = frame.address;
  frame.raw_bytes[j++] = frame.control;
  frame.raw_bytes[j++] = frame.bcc1;
  for (int k = 0; k < frame.data_size; k++) {
    frame.raw_bytes[j++] = frame.data[k];
  }
  /* BCC2 stuffing*/
  if (bcc == ESCAPE) {
    frame.raw_bytes[j++] = ESCAPE;
    frame.raw_bytes[j++] = ESCAPE_ESC;
  } else if (bcc == FLAG) {
    frame.raw_bytes[j++] = ESCAPE;
    frame.raw_bytes[j++] = ESCAPE_FLAG;
  } else
    frame.raw_bytes[j++] = bcc;

  frame.raw_bytes[j++] = FLAG;

  // ! remove next comment if you want to see the coded message being
written
  // print_message(frame, TRUE);
  conta = 1;
  int count = -1;

  do {
    if ((count = write(fd, frame.raw_bytes, j)) != ERROR) {
      printf("Message sent! Waiting for ACK\n");
    } else {
```

```
      printf("Message not sent!\n");
      return ERROR;
      // adicionei esta linha, pq caso não escreva corretamente
      // deve retornar -1 para escrever de novo
    }
    alarm(TIMEOUT);
    flag = 0;

    unsigned char ack = receive_acknowledgement(fd);
    if (ack == C_REJ0 || ack == C_REJ1) {
      printf("Received negative ACK\n");
      alarm(RESET_ALARM);
      return ERROR;
    }
    // Retransmition
    if ((ack == C_RR0 && current_frame == 0) ||
        (ack == C_RR1 && current_frame == 1)) {
      printf("Received positive ACK (retransmition)\n");
      alarm(RESET_ALARM);
      // returns error but to the application only means it has to
      // send the same frame again
      return ERROR;
    }

    if ((ack == C_RR0 && current_frame == 1) ||
        (ack == C_RR1 && current_frame == 0)) {
      printf("Received positive ACK\n");
      alarm(RESET_ALARM);
      current_frame = (current_frame == 0) ? 1 : 0; // changes the
current frame
      return count;
    } else {
      printf("Timed out\nTrying again\n");
      alarm(RESET_ALARM);
    }
    printf("Couldn't receive ACK in time\n");
  } while (flag && conta < 4);

  return ERROR;
}

int llread(int fd, char *buffer) {
  information_frame_t information_frame;
  information_frame.raw_bytes = (unsigned char *)malloc(sizeof(unsigned
char));

  int i = 0;
  int part = 0;
  unsigned char rcv_msg;
  printf("Reading...\n");

  // * lógica: processar os dados todos em raw bytes, depois fazer o
unstuffing,
  // * e depois fazer o tratamento dos dados

  /*
    part 0 - before first flag
    part 1 - between flag start and flag stop
    part 2 - after flag stop */
  while (part != 2) {
```

```
    read(fd, &rcv_msg, 1);

    if (rcv_msg == FLAG && part == 0) {
      part = 1;
      continue;
    } else if (rcv_msg == FLAG && part == 1) {
      part = 2;
      break;
    }
    information_frame.raw_bytes[i++] = rcv_msg;
    information_frame.raw_bytes = (unsigned char
*)realloc(information_frame.raw_bytes, (i + 1));
  }

  int data_size = i;
  /* UNSTUFFING BYTES */
  int j = 0, p = 0;
  for (; j < i && p < i; j++) {
    if (information_frame.raw_bytes[p] == ESCAPE) {
      information_frame.raw_bytes =
          (unsigned char *)realloc(information_frame.raw_bytes, --
data_size);
      if (information_frame.raw_bytes[p + 1] == ESCAPE_ESC)
        information_frame.raw_bytes[j] = ESCAPE;
      else if (information_frame.raw_bytes[p + 1] == ESCAPE_FLAG)
        information_frame.raw_bytes[j] = FLAG;
      p += 2;
    } else {
      information_frame.raw_bytes[j] = information_frame.raw_bytes[p];
      p++;
    }
  }

  information_frame.data =
      (unsigned char *)malloc((data_size - 4) * sizeof(unsigned char));

  information_frame.address = information_frame.raw_bytes[0];
  information_frame.control = information_frame.raw_bytes[1];
  information_frame.bcc1 = information_frame.raw_bytes[2];
  p = 0;
  for (int byte = 3; byte < data_size - 1; byte++) {
    information_frame.data[p++] = information_frame.raw_bytes[byte];
  }
  information_frame.bcc2 = information_frame.raw_bytes[data_size - 1];
  information_frame.data_size = data_size - 4;

  // ! remove *sleep* comments if you want to check what happens when ACK
is not received in time
  // ! remove print_message comment if you want to see the data byte-by-
byte
  int bccError = verify_message(information_frame);
  if (bccError == ERROR) {
    // sleep(15);
    send_acknowledgement(fd, current_frame, FALSE);
  } else {
    // sleep(4);
    send_acknowledgement(fd, current_frame, TRUE);
    current_frame = (current_frame == 0) ? 1 : 0;
    // print_message(information_frame, FALSE);
  }
```

```
  for (i = 0; i < information_frame.data_size; i++) {
    buffer[i] = information_frame.data[i];
  }

  free(information_frame.raw_bytes);
  free(information_frame.data);
  return (bccError == OK) ? information_frame.data_size : ERROR;
}
```

data_link.h

```
#include "utils.h"

#define TRANSMITTER 0
#define RECEIVER 1
#define STOP_AND_WAIT 50

/**
 * current I-Frame
 */
static int current_frame = 0;

/**
* takes file descriptor (port) and sends a code msg in a
* supervision frame
*/
int send_supervision_frame(int fd, unsigned char msg, unsigned char
address);

/**
* receives a supervision frame with controll as msg
*/
int receive_supervision_frame(int fd, unsigned char msg);

/**
* receives a ACK frame and returns it's control byte
*/
unsigned char receive_acknowledgement(int fd);

int send_acknowledgement(int fd, int frame, int accept);

/**
 * Reading Fucntion
 * @param fd Serial Port to be read
 */
int receive_set(int fd);

/**
 * This function sends a SET Control frame and expects an UA
 */
int send_set(int fd);

int disconnect_writer(int fd);

int disconnect_reader(int fd);

/**
 * This function opens a port and returns the file descriptor
 * @param port port number to be open
```

```
 * @param type RECEIVER or TRANSMITTOR
 */
int llopen(int port, int type);

/**
 * Same as llopen but this one closes the fd
 */
int llclose(int fd, int type);

/**
 * @brief Function to write a buffer to a file descriptor
 * This function takes the buffer and sends it trought the port,
 * after the byte-stuffing
 * @returns -1 if error or number of bytes written for success
 */
int llwrite(int fd, char *buffer, int length);

/**
 * @brief Function to read a buffer from a file descriptor
 * This function reads a buffer from the port and returns the number
 * @returns -1 if error or number of bytes read for success
 */
int llread(int fd, char *buffer);
```

files.c

```
#include "files.h"

unsigned long get_file_size(FILE* f) {
  fseek(f, 0, SEEK_END); // seek to end of file
  unsigned long size = ftell(f); // get current file pointer
  fseek(f, 0, SEEK_SET); // seek back to beginning of file
  // proceed with allocating memory and reading the file
  return size;
}

unsigned char* read_file(FILE* f, unsigned long filesize) {
  unsigned char* data = (unsigned char*) malloc (filesize);
  fread(data, sizeof(unsigned char), filesize, f);
  return data;
}

unsigned char* split_file(unsigned char* data, unsigned long index_start,
unsigned long index_end) {
  int range = index_end - index_start + 1;
  unsigned char* frame = (unsigned char*) malloc (range);

  for (int k = 0; k < range; k++) {
    frame[k] = data[index_start + k];
  }

  return frame;
}

void join_file(unsigned char* data, unsigned char* frame, unsigned long
size, int index) {
  for (int j = 0; j < size; j++) {
    data[index + j] = frame[j];
  }
}
```

```
void write_file(char* name, unsigned char* bytes, unsigned long size) {
  FILE *fh = fopen (name, "wb");
  if (fh != NULL) {
      fwrite (bytes, sizeof (unsigned char), size, fh);
      fclose (fh);
  }
}int fd, char *buffer);
```

files.h

```
#include "macros.h"

unsigned long get_file_size(FILE* f);

unsigned char* read_file(FILE* f, unsigned long filesize);

unsigned char* split_file(unsigned char* data, unsigned long index_start,
unsigned long index_end);

void join_file(unsigned char* data, unsigned char* frame, unsigned long
size, int index);

void write_file(char* name, unsigned char* bytes, unsigned long size);
```

macros.h

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

#define BAUDRATE        B38400
#define MODEMDEVICE     "/dev/ttyS1"
#define _POSIX_SOURCE   1 /* POSIX compliant source */
#define FALSE           0
#define TRUE            1
#define OK              0
#define ERROR           -1
#define TRIES           3
#define TIMEOUT         3
#define RESET_ALARM     0

// Mesh Macros
#define FLAG            0x7E
#define A_3             0x03
#define A_1             0x01
#define SET             0x03
#define UA              0x07
#define DISC            0x0B
#define SET_BCC         A ^ SET
#define UA_BCC          A ^ UA
```

```
// Control Macros (there is another way of defining them)
#define C_I0            0x00
#define C_I1            0x40
#define C_RR0           0x05
#define C_RR1           0x85
#define C_REJ0          0x01
#define C_REJ1          0x81

//Byte Stuffing
#define ESCAPE          0x7D
#define ESCAPE_ESC      0x5D
#define ESCAPE_FLAG     0x5E

// packet Macros
#define DATA            0x1
#define START           0x2
#define STOP            0x3
#define FILE_SIZE       0x0
#define FILE_NAME       0x1
#define PACKET_SIZE     1024
```

noncanonical.c

```
/*Non-Canonical Input Processing*/
#include "noncanonical.h"

extern int reetransmit;
static struct termios oldtio;
static struct termios newtio;

int open_reader(char *port) {
  int fd;

  /* top layer does the verification of the port name */

  /*
    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(port, O_RDWR | O_NOCTTY);
  if (fd < 0) {
    perror(port);
    return -1;
  }

  if (tcgetattr(fd, &oldtio) == -1) { /* save current port settings */
    perror("tcgetattr");
    return -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] = 1; /* inter-character timer unused */
  newtio.c_cc[VMIN] = 0;  /* blocking read until 1 chars received */

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

  tcflush(fd, TCIOFLUSH);

  if (tcsetattr(fd, TCSANOW, &newtio) == -1) {
    perror("tcsetattr");
    return -1;
  }
  printf("New termios structure set\n");

  return fd;
}

int close_reader(int fd) {
  if (tcsetattr(fd, TCSANOW, &oldtio) == -1) {
    perror("tsetattr");
    return -1;
  }
  close(fd);
  return 0;
}
```

noncanonical.h

```
#include "macros.h"

/**
 * @brief Opens the reader
 */
int open_reader(char *port);

/**
 * @brief Reset and close the reader
 */
int close_reader(int fd);
```

utils.c

```
#include "utils.h"

control_packet_t parse_control_packet(unsigned char *raw_bytes, int size)
{
  control_packet_t packet;
  memset(&packet, 0, sizeof(control_packet_t));
  packet.control = raw_bytes[0];

  char *name;
  int namesize = 0;

  unsigned char* filesize;
  int filesize_size = 0;

  for (int i = 1; i < size; i++) {
```

```
    if (raw_bytes[i] == FILE_SIZE) {
      int length = raw_bytes[++i];
      int offset = i + length;
      filesize = (unsigned char*) malloc (length);
      for (int k = 0; i < offset; k++) {
        filesize[k] = raw_bytes[++i];
        filesize_size++;
      }
      continue;
    }
    if (raw_bytes[i] == FILE_NAME) {
      int length = raw_bytes[++i];
      name = (unsigned char *) malloc (length);
      int offset = i + length;
      for (int j = 0; i < offset;) {
        name[j++] = raw_bytes[++i];
        namesize++;
      }
      continue;
    }
  }

  packet.file_name = (unsigned char*) malloc (namesize + 1);
  memcpy(packet.file_name, name, namesize);
  packet.file_name[namesize] = '\0';
  free(name);

  packet.filesize_size = filesize_size;
  packet.file_size = (unsigned char*) malloc (filesize_size);
  memcpy(packet.file_size, filesize, filesize_size);
  free(filesize);

  return packet;
}

data_packet_t parse_data_packet(unsigned char *raw_bytes, int size) {
  data_packet_t packet;
  memset(&packet, 0, sizeof(data_packet_t));
  packet.raw_bytes_size = size;
  packet.control = raw_bytes[0];
  packet.sequence = raw_bytes[1];

  packet.data_field_size = (raw_bytes[2] << 8) | raw_bytes[3];

  for (int i = 0; i < packet.data_field_size; i++) {
    packet.data[i] = raw_bytes[4 + i];
  }

  return packet;
}

void print_control_packet(control_packet_t packet) {
  printf("---- CONTROL PACKET ----\n");
  switch (packet.control) {
  case START:
    printf("Control: START (0x%x)\n", packet.control);
    break;
  case STOP:
    printf("Control: STOP (0x%x)\n", packet.control);
    break;
```

```
  default:
    break;
  }

  printf("Size: %ld %#lx\n", array_to_number(packet.file_size,
packet.filesize_size), array_to_number(packet.file_size,
packet.filesize_size));
  printf("Name: %s\n", packet.file_name);
  printf("-------------------------\n");
}

void print_data_packet(data_packet_t* packet, int full_info) {
  printf("---- DATA PACKET ----\n");
  printf("Control: - (0x%x)\n", packet->control);
  printf("Data size: %d (0x%x)\n", packet->data_field_size,
         packet->data_field_size);
  printf("Sequence: %d (0x%x)\n", packet->sequence, packet->sequence);

  if (full_info) {
    for (int i = 0; i < packet->data_field_size; i++) {
      printf("DATA[%d]: %c (0x%x)\n", i, packet->data[i], packet-
>data[i]);
    }
  }

  printf("----------------------\n");
}

void print_message(information_frame_t frame, int stuffed) {
  printf("Address: 0x%x\n", frame.address);
  printf("Control: 0x%x\n", frame.control);
  printf("BCC1: 0x%x\n", frame.bcc1);
  int j = 0;
  for (int i = 0; i < frame.data_size; i++) {
    if (frame.data[i] == ESCAPE && stuffed) {
      printf("DATA[%d]: 0x%x - ESCAPE\n", j++, frame.data[i++]);
      if (frame.data[i] == ESCAPE_ESC) {
        printf("DATA[%d]: 0x%x - ESCAPED ESCAPE\n", j++, frame.data[i]);
      } else if (frame.data[i] == ESCAPE_FLAG) {
        printf("DATA[%d]: 0x%x - ESCAPED FLAG\n", j++, frame.data[i]);
      }
    } else {
      printf("DATA[%d]: 0x%x - %c (char)\n", j++, frame.data[i],
frame.data[i]);
    }
  }
  printf("BCC2: 0x%x\n", frame.bcc2);
  printf("Message: %s - size: %d - strlen: %ld\n", frame.data,
frame.data_size,
         strlen(frame.data));
}

int verify_message(information_frame_t frame) {
  if (frame.bcc1 != (frame.control ^ frame.address)) {
    return ERROR;
  }
  unsigned char bcc2 = 0xff;
  for (int i = 0; i < frame.data_size; i++) {
    bcc2 = frame.data[i] ^ bcc2;
  }
```

```
  if (bcc2 != frame.bcc2) {
    return ERROR;
  }

  return OK;
}

void print_elapsed_time(struct timespec start) {
  struct timespec end;
  clock_gettime(CLOCK_MONOTONIC_RAW, &end);
  double delta = (end.tv_sec - start.tv_sec) * 1000.0 +
                 (end.tv_nsec - start.tv_nsec) / 1000000.0;
  printf("Elapsed time: %f s\n\n", delta);
}

int check_connection(int fd) {
  if (fcntl(fd, F_GETFD) == -1) {
    printf("Connection closed\n");
    return -1;
  }
  return 0;
}

unsigned long array_to_number(unsigned char* buffer, unsigned int size) {
  unsigned long value = 0;
  int offset = 0;
  for (int i = 0; i < size; i++) {
    value |= buffer[i] << (8 * offset++);
  }

  return value;
}

unsigned int number_to_array(unsigned long num, unsigned char* buffer) {
  unsigned int size = 0;

  for (int i = 0; i < sizeof(unsigned long); i++) {
    buffer[i] = (num >> (8 * i)) & 0xff;
    size += 1;
  }
  for (int i = sizeof(unsigned long) - 1; i != 0; i--) {
    if (buffer[i] != 0) break;
    size--;
  }

  return size;
}
```

utils.h

```
#include "macros.h"
#include "writenoncanonical.h"
#include "noncanonical.h"
#include "app_structs.h"

/**
 * @brief This function parses a buffer of raw bytes to a
control_packet_t
 * structure
```

```
 * This method takes the raw_bytes and parses them into a control packet
 */
control_packet_t parse_control_packet(unsigned char *raw_bytes, int
size);

/**
 * @brief This function parses a buffer of raw bytes to a data_packet_t
 * structure
 * This method takes the raw_bytes and parses them into a data packet
 */
data_packet_t parse_data_packet(unsigned char *raw_bytes, int size);

/**
 * @brief This function pretty-prints a control packet
 */
void print_control_packet(control_packet_t packet);

/**
 * @brief This function pretty-prints a data packet
 */
void print_data_packet(data_packet_t* packet, int full_info);

/**
 * @brief Method to pretty-print the elapsed time between two frames
 */
void print_elapsed_time(struct timespec start);

/**
 * @brief Method to verify an I-Frame
 * Checks if there are errors on the BCC bytes
 * @returns 0 if no error or -1 for error
 */
int verify_message(information_frame_t frame);

/**
 * @brief Method to pretty-print an information frame details
 */
void print_message(information_frame_t frame, int coded);

/**
 * @brief Takes fd and checks if it hasn't closed
 */
int check_connection(int fd);

/**
 * @brief takes an array with length size, and converts it to an 8byte
number
 * array[0] = MSB ; array[size - 1] = LSB
 */
unsigned long array_to_number(unsigned char* buffer, unsigned int size);

/**
 * @brief Method to convert a 8byte number into an 8 byte char array
 * array[0] = MSB ; array[return - 1] = LSB
 * @return array's size
 */
unsigned int number_to_array(unsigned long num, unsigned char* buffer);
```

writenoncanonical.c

```
/*Non-Canonical Input Processing*/
#include "writenoncanonical.h"

int flag = 1, conta = 1;
extern int reetransmit;
static struct termios oldtio;
static struct termios newtio;

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

int open_writer(char *port) {
  /* top level layer should verifiy ports name */

  /*
          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.
  */

  int fd = open(port, O_RDWR | O_NOCTTY);
  if (fd < 0) {
    perror(port);
    return -1;
  }

  if (tcgetattr(fd, &oldtio) == -1) {
    /* save current port settings */
    perror("tcgetattr");
    return -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] = TIMEOUT; /* inter-character timer unused */
  newtio.c_cc[VMIN] = 0;        /* blocking read 1 char received */

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

  tcflush(fd, TCIOFLUSH);

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

```
  }
  /*struct sigaction action;
       sigemptyset(&action.sa_mask);
       action.sa_handler = atende;
       action.sa_flags = 0;
       sigaction(SIGALRM,&action,NULL);*/ // instala  rotina que atende
interrupcao
  signal(SIGALRM, atende);
  siginterrupt(SIGALRM, 1);
  printf("New termios structure set\n");

  return fd;
}

int close_writer(int fd) {
  if (tcsetattr(fd, TCSANOW, &oldtio) == -1) {
    perror("tcsetattr");
    return -1;
  }

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

writenoncanonical.h

```
#include "macros.h"

/**
* function to open and set port
*/
int open_writer(char *port);

/**
* function to reset and close port
*/
int close_writer(int fd);
```