U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Relatório

1º Trabalho Laboratorial

Mestrado Integrado em Engenharia Informática e Computação

Redes de Computadores

6 de novembro de 2018

3MIEIC06

Miguel Dias de Carvalho	**up201605757**@fe.up.pt
Sandro Miguel Tavares Campos	**up201605947**@fe.up.pt
Tiago Pinho Cardoso	**up201605762**@fe.up.pt

Índice

Sumário

O presente relatório, realizado no âmbito da unidade curricular de Redes de Computadores, aborda a realização de um trabalho laboratorial com a finalidade de transferir ficheiros entre máquinas distintas conectadas por porta de série.

Tendo este sido concluído com sucesso, e observando a plena operacionalidade do programa pretendido, proceder-se-á a uma explicação mais detalhada acerca da sua implementação.

Introdução

O objetivo deste trabalho é a implementação de um protocolo de ligação de dados que permita ser testado num ambiente laboratorial com dois sistemas conectados através de porta de série. Com este relatório pretende-se efetuar uma análise mais aprofundada do que está na base da sua implementação, assim como realizar um estudo da sua eficiência. Segue-se a sua estrutura:

- **Arquitetura -** apresentação dos blocos funcionais e suas interfaces.

- **Estrutura do código -** exposição da API, principais funções, estruturas de dados e a sua relação com a arquitetura escolhida.

- **Casos de uso principais -** identificação e demonstração das sequências de chamada de funções.

- **Protocolo de ligação lógica -** identificação dos principais aspetos funcionais e descrição da estratégia de implementação ilustrando-a através de extratos de código.

- **Validação -** descrição dos testes efetuados ao programa.

- **Eficiência do protocolo de ligação de dados -** caraterização e análise estatística da eficiência do protocolo de ligação de dados realizada com recurso ao código desenvolvido.

- **Conclusões -** síntese da informação apresentada e reflexão sobre os objetivos de aprendizagem alcançados.

Arquitetura

Os blocos funcionais presentes neste trabalho são dois, um referente ao transmissor de informação, e o outro referente ao recetor dessa mesma informação. Por sua vez, as interfaces da camada de ligação de dados e da camada da aplicação encontram-se implementadas de forma independente, não só no interior de cada um destes blocos funcionais, mas também entre cada um destes.

Estrutura do código

De acordo com o modelo arquitetural referido, seguem-se as principais funções e estruturas de dados usadas por cada um dos blocos funcionais especificados:

Writer

a) Camada da ligação de dados

Principais funções

- **llopen:** envio da trama SET e receção da trama UA, para estabelecimento da ligação lógica.
- **llclose:** envio da trama DISC, e envio da trama UA após receção de uma trama DISC do recetor, para encerramento da ligação lógica.
- **llwrite:** envio de tramas de informação numeradas após estas terem sido submetidas ao mecanismo de *stuffing.*

Principais estruturas de dados

```
typedef struct
{
    int frame;                    // current frame number
    int timeout;                  // timeout flag
    int nTries;                   // number of timeout tries
    struct termios oldtio, newtio; // structs storing terminal configurations
} DataLink;
```

b) Camada da aplicação

Principais funções

- **sendFile:** abertura do ficheiro pretendido, desencadeando todo o processo de formação e envio de pacotes através da invocação de outras funções da mesma camada.
- **getControlpackage:** inicialização dos pacotes de controlo inicial e final.
- **getBufPackage:** extração de informação proveniente do *buffer* do ficheiro.
- **getDataPackage:** constituição de pacotes de dados.

Principais estruturas de dados

```
typedef struct
{
    int fd;            // serial port descriptor
    char *filename;    // name of file to be sent
    int package;       // number of current package
    int totalPackages; // total number of packages
} Application;
```

Reader

a) Camada da ligação de dados

Principais funções

- **llopen:** receção da trama SET e envio da trama UA, para estabelecimento da ligação lógica.
- **llclose:** receção e envio da trama DISC, seguido da receção da trama UA, para encerramento da ligação lógica.
- **llread:** receção de tramas de informação numeradas, realizando o seu *destuffing*.

Principais estruturas de dados

```
typedef struct
{
    int frame;                    // current frame number
    int expectedFrame;            // expected frame number
    struct termios oldtio, newtio; // structs storing terminal configurations
} DataLink;
```

b) Camada da aplicação

Principais funções

- **receiveFile:** alocação e preenchimento do *buffer* com a informação recebida do emissor.
- **parseStartPackage:** efetua a extração da informação do tamanho e nome do ficheiro.
- **parseDataPackage:** realiza a extração dos dados provenientes no pacote de dados.

Principais estruturas de dados

```
typedef struct
{
    int fd;                  // serial port descriptor
    unsigned char *filename; // name of received file
    off_t filesize;          // filesize to be received
    int package;             // current package
    int totalPackages;       // total number of packages
} Application;
```

Casos de uso principais

De forma a interagir com o programa o utilizador deve recorrer à interface existente da linha de comandos, caso de uso que é único.

Para dar início à aplicação, e por sua vez ao envio do ficheiro do lado emissor, deve proceder-se à inserção de dois argumentos ao invocar o executável *writer*. O primeiro indica a porta de série a ser utilizada, no formato **/dev/ttySX**, e o segundo, o ficheiro a ser enviado para o programa recetor. Do lado recetor, o programa deve ser iniciado apenas com o argumento da porta de série a realizar a transferência.

Exemplo: ./writer /dev/ttyS0 penguin.gif e ./reader /dev/ttyS0

Sequência de execução do programa

1. Configuração da porta de série a realizar a transferência.
2. Estabelecimento da ligação lógica.
3. Abertura do ficheiro a enviar.
4. Envio de dados, pelo emissor.
5. Receção de dados e a sua escrita em buffer, pelo recetor.
6. Fecho do ficheiro a enviar.
7. Criação do ficheiro, pelo recetor, com os dados recebidos.
8. Encerramento da ligação lógica.

Protocolo de ligação lógica

Com o objetivo de fornecer um serviço de comunicação fiável entre os dois sistemas ligados por porta de série, implementou-se um protocolo de ligação lógica para transmissão assíncrona, baseada no mecanismo *Stop&Wait*.

Principais aspetos funcionais

- Configuração da porta de série, fazendo *backup* das configurações originais.
- Estabelecimento e encerramento da ligação lógica.
- Realização dos processos de *framing*, de *stuffing* e *destuffing* dos pacotes recebidos.
- Envio de tramas de controlo e de informação.
- Recuperação de erros que surjam durante a transferência dos dados.

```
int llopen(int fd);
```

llopen tem a função de estabelecer a ligação lógica, sendo para isso necessário, inicialmente, alterar a configuração da porta de série escolhida.

O emissor usa-a em seguida para preparar e enviar a trama **SET**, ativar o alarme com uma duração de ***timeout*** predefinida, e aguardar posteriormente uma resposta **UA** de confirmação a ser enviada pelo recetor, e que é processada pela função *readControlFrame*. No caso de o alarme ser desencadeado, a trama é enviada até que se esgote o limite também definido para o número de tentativas, **nTries**. O estabelecimento da ligação falha e a função retorna -1, caso o número de tentativas seja excedido, ou é bem-sucedido e retorna 1 caso contrário.

```
int llclose(int fd);
```

llclose tem a função de encerrar a ligação lógica, repondo a configuração da porta de série escolhida para a sua configuração original.

O emissor usa-a para preparar e enviar a trama **DISC** e aguardar posteriormente uma resposta **DISC** semelhante a ser enviada pelo recetor. A receção desta trama está igualmente dependente de um alarme, e o encerramento da ligação dá-se após o envio de uma trama **UA**, por parte do emissor, confirmando a sua intenção de concluir a desconexão. A função termina com sucesso retornando 1, ou -1 caso contrário.

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

llwrite é responsável pelo envio das tramas para os *buffers* da porta de série.

Esta invoca a função *stuff* para realizar o ***stuffing*** do pacote de dados recebido, efetuando de igual modo o ***framing*** ao conteúdo resultante. Este é o processo necessário à criação da designada **trama de informação** que está pronta a enviar.

O envio da trama está envolto no mesmo mecanismo de retransmissão referido anteriormente nas funções *llopen* e *llclose*, sendo que a resposta aguardada pelo emissor é desta vez **RR** ou **REJ**, indicando a correta receção, ou a necessidade do reenvio da trama, respetivamente.

```
unsigned char *llread(int fd, int *frameSize);
```

llread é responsável pela receção das tramas a partir dos *buffers* da porta de série.

O recetor usa-a para caráter a caráter, efetuar a leitura da trama de informação enviada. Durante este procedimento realiza-se o seu ***destuffing*** e verifica-se se o parâmetro **BCC2** da trama é o correto, isto é, se coincide à operação de XOR entre todos os bytes correspondentes

aos dados para essa trama. Caso isso aconteça, a resposta enviada ao recetor é a de confirmação **RR** e o buffer de dados é retornado, caso contrário a resposta é de rejeição **REJ** e o seu retorno é NULL.

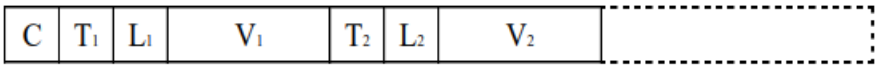
Protocolo de aplicação

Principais aspetos funcionais

- Criação dos pacotes de controlo e de informação.
- Fragmentação dos dados, pelo emissor, e a sua agregação, por parte do recetor.
- Leitura do ficheiro a enviar, pelo emissor, e a sua criação, por parte do recetor.

```
unsigned char *getControlPackage(unsigned char c, off_t filesize,
char *filename, int *packageSize);
```

getControlPackage permite constituir o pacote de controlo a enviar, definido pelo parâmetro ***c***, que indica o início ou o fim da transferência. Nos bytes que se seguem é colocada a restante informação relativa ao ficheiro, ***filesize*** e ***filename***, no formato TLV (*Type*, *Length*, *Value*). Neste formato especifica-se o tipo, tamanho e valor do parâmetro a representar. A função que efetua o processo inverso, do lado recetor, é a ***parseStartPackage***.

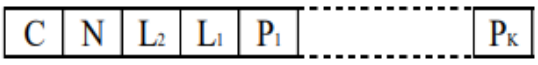



```
unsigned char *getBufPackage(unsigned char *filebuf, off_t *idx, int
*packageSize, off_t filesize);
```

getBufPackage retorna um segmento de dados de tamanho ***packageSize***, a partir de um determinado índice ***idx*** do *buffer* contendo a totalidade do conteúdo do ficheiro.

```
unsigned char *getDataPackage(Application *app, unsigned char *buf,
int *packageSize, off_t filesize);
```

getDataPackage possibilita a obtenção de um pacote de dados formado a partir de um *segmento* de dados, ***buf***. Este sofre um processo de ***framing*** em que se lhe é colocado um cabeçalho, contendo parâmetros relativos ao seu número de sequência e ao tamanho do pacote. Este cabeçalho preenche os quatro bytes iniciais enquanto que os dados seguem na secção P1 a Pk, representada na figura abaixo. A função que realiza o processo inverso e que guarda os dados, do lado recetor, é a ***parseDataPackage***.



Validação

A fim de verificar a boa operacionalidade e a robustez do programa desenvolvido, este foi sujeito a uma bateria de testes relevantes, e que foram concluídos com sucesso:

- Envio de ficheiros com extensões distintas e de tamanhos variados;
- Envio de ficheiros com uma temporária interrupção da ligação durante a transmissão;
- Envio de ficheiros introduzindo erros com o auxílio de um cabo de cobre;
- Envio de ficheiros fazendo variar o tamanho dos pacotes;
- Envio de ficheiros fazendo variar a *baudrate*.

Eficiência do protocolo de ligação de dados

O protocolo de ligação de dados implementado tem por base o protocolo teórico de ***Stop&Wait*** em que ocorre a transmissão assíncrona de *packets* de informação entre um emissor e um recetor. Durante este processo, e para cada *packet* enviado, o emissor requer o envio de uma mensagem de confirmação, **ACK**, por parte do recetor. O recetor, ao receber o packet sem erros, confirma essa receção com o envio de um **ACK**, ou **NACK**, caso contrário. Uma vez recebida uma resposta positiva do recetor, o emissor continua a transmissão com o envio do próximo *packet*, ou inicia a retransmissão para o *packet* cuja receção falhou. Além disso, e para evitar que as respostas se percam indefinidamente, é aconselhada a implementação de um mecanismo de *timeout*.

Posto isto, na aplicação desenvolvida, as respostas passíveis de ocorrer são **RR** e **REJ** associadas ao número de sequência, 0 ou 1, da trama em questão. Esta numeração permite saber que trama enviar e evitar o tratamento de tramas enviadas em duplicado.

A eficiência do protocolo de ligação de dados pode por sua vez ser analisada de diversas perspetivas. No anexo nº II podemos observar os dados que nos permitiram inferir as conclusões que apresentamos a seguir.

Eficiência com a variação do tamanho das tramas de informação

Verifica-se, tal como é previsto teoricamente, que quanto maior o tamanho de cada pacote enviado, mais eficiente tende a ser o protocolo. Isto justifica-se com o facto de quanto maior a trama, menor o número de tramas a enviar, para uma determinada quantidade de informação.

Eficiência com a variação da capacidade da ligação

Quanto maior a capacidade da ligação, menor é a eficiência do protocolo.

Eficiência com variação do *frame error ratio* (FER)

O *frame error ratio* é provocado por erros que envolvem os *BCC* e que têm como consequência a diminuição da eficiência do protocolo.

Aquando da receção do *BCC1*, este ou está correto para que continue o processamento da trama, ou se retorna a um estado anterior aguardando uma nova trama, que possua um parâmetro *BCC1* válido. Por outro lado, no caso de ocorrerem erros no *BCC2*, não só estes são detetados de forma imediata como um pedido de reenvio dessa mesma trama é efetuado.

De acordo com este raciocínio é intuitivo que os erros no *BCC1* sejam os que mais impacto têm na eficiência do protocolo.

Conclusões

A implementação do protocolo de ligação de dados proposto permitiu a constituição de um serviço de comunicação fiável entre sistemas ligados por cabo de série. O conjunto de fundamentos teóricos necessário é intuitivo e possibilitou uma maior compreensão do seu funcionamento.

Um dos conceitos de maior importância neste tipo de projeto, o de **independência entre camadas**, demonstra a relevância de isolar as camadas de ligação de dados e de aplicação. Assim, a camada de aplicação não necessita de conhecer o modo de funcionamento da camada de ligação de dados, mas apenas de que forma aceder ao serviço por ela prestado.

De acordo com o guião fornecido consideramos que os objetivos deste trabalho laboratorial foram alcançados, dando-o, portanto, por bem-sucedido.

Anexo I – código fonte

writer.h

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

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

#define TIMEOUT 3   // number of seconds for timeout
#define MAX_TRIES 3 // number of maximum timeout tries

// ESCAPE flag
#define ESCAPE 0x7d

// data package flags
#define C_START 0x02 // control package start state byte
#define C_END 0x03   // control package end state byte

// control package flags
#define DATAFRAME_C 0x01
#define T1 0x00 // filesize parameter type
#define L1 0x04 // filesize 4 octets
#define T2 0x01 // filepath parameter type

// information frame flags
#define PACKAGE_SIZE 100
#define C_0 0x00
#define C_1 0x40

// control frame flags
#define FLAG 0x7E
#define A_TRANSMITTER 0x03
```

```c
#define A_RECEIVER 0x01
#define C_SET 0x03
#define C_DISC 0x0B
#define C_UA 0x07
#define C_RR0 0x05
#define C_RR1 0x85
#define C_REJ0 0x01
#define C_REJ1 0x81

// BCC1 probability error
#define BCC_PE 0
#define BCC2_PE 0

// progress bar default dimension
#define PROGRESS_BAR_DIM 20

typedef enum
{
    INITIAL,
    STATE_FLAG,
    STATE_A,
    STATE_C,
    STATE_BCC
} State;

typedef struct
{
    int fd;            // serial port descriptor
    char *filename;    // path of file to be sent
    int package;       // number of current package
    int totalPackages; // total number of packages
} Application;

typedef struct
{
    int frame;                    // current frame number
    int timeout;                  // timeout flag
    int nTries;                   // number of timeout tries
    struct termios oldtio, newtio; // structs storing terminal configurations
} DataLink;

// --------------- APPLICATION LAYER --------------------

Application *initApplicationLayer(char *port, char *filepath);

void destroyApplicationLayer(Application *app);
```

```
int sendFile(Application *app);

unsigned char *getControlPackage(unsigned char c, off_t filesize, char *filepath,
int *packagesize);

int sendControlPackage(int fd, int c, off_t filesize, char *filepath);

unsigned char *getBufPackage(unsigned char *filebuf, off_t *idx, int *packageSize,
off_t filesize);

unsigned char *getDataPackage(Application *app, unsigned char *buf, int
*packagesize, off_t filesize);

int sendDataPackage(Application *app, unsigned char *buf, int packagesize, off_t
filesize);

// ----------- END OF APPLICATION LAYER -----------------

// ---------------- DATA LINK LAYER ---------------------

DataLink *initDataLinkLayer();

void destroyDataLinkLayer();

unsigned char *readControlFrame(int fd, unsigned char c);

unsigned char readControlC(int fd);

int llopen(int fd);

int llclose(int fd);

void stuff(unsigned char *buf, int *bufsize);

int llwrite(int fd, unsigned char *buf, int bufsize);

// -------------- END OF DATA LINK LAYER ---------------

// -------------------- UTILS -------------------------

void timeoutHandler(int signo);

void installAlarm();

void uninstallAlarm();
```

```
unsigned char *errorBCC(unsigned char *package, int sizePackage);

unsigned char *errorBCC2(unsigned char *package, int packageSize);

void printProgressBar(Application *app);

void printArr(unsigned char arr[], int size);

// ------------------ END OF UTILS ----------------------
```

writer.c

```
#include "writer.h"

int DEBUG = FALSE;
int received = FALSE;

DataLink *dl;

int main(int argc, char **argv)
{
  struct timespec start, end;

  if ((argc < 3) ||
      ((strcmp("/dev/ttyS0", argv[1]) != 0) &&
       (strcmp("/dev/ttyS1", argv[1]) != 0)))
  {
    printf("Usage:\tnserial SerialPort File\n\tex: nserial /dev/ttyS1
Penguin.gif\n");
    exit(1);
  }

  // initialize application layer
  Application *app = initApplicationLayer(argv[1], argv[2]);

  // install alarm handler
  installAlarm();

  // start transfer timing
  clock_gettime(CLOCK_REALTIME, &start);

  // open connection
  if (!llopen(app->fd))
  {
```

```
    printf("[llopen] - failed to open connection!\n");
    return -1;
  }

  // opens file to transmit
  sendFile(app);

  // close connection
  if (!llclose(app->fd))
  {
    printf("[llclose] - failed to close connection!\n");
    return -1;
  }

  // stop transfer timing
  clock_gettime(CLOCK_REALTIME, &end);

  double elapsed = (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec) /
1E9;
  printf("# File transferred in %.2f seconds!\n", elapsed);

  uninstallAlarm();

  // destroy application layer
  destroyApplicationLayer(app);

  return 0;
}

// --------------- APPLICATION LAYER --------------------
Application *initApplicationLayer(char *port, char *filename)
{
  Application *app = (Application *)malloc(sizeof(Application));

  /*
  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.
  */
  app->fd = open(port, O_RDWR | O_NOCTTY);
  if (app->fd < 0)
  {
    perror(port);
    exit(-1);
  }

  // store file name
```

```
  app->filename = filename;

  // start package numeration
  app->package = 0;

  // initialize datalink layer
  dl = initDataLinkLayer();

  return app;
}

void destroyApplicationLayer(Application *app)
{
  destroyDataLinkLayer();

  // free application layer
  free(app);
}

int sendFile(Application *app)
{
  // open file
  FILE *file = fopen(app->filename, "rb");
  if (file == NULL)
  {
    perror("[openfile] - could not read file!\n");
    exit(-1);
  }

  // extract file metadata
  struct stat metadata;
  stat((char *)app->filename, &metadata);
  off_t filesize = metadata.st_size;

  if (DEBUG)
    printf("[sendFile] - file %s of size %ld bytes.\n", app->filename, filesize);

  app->totalPackages = filesize / PACKAGE_SIZE;
  if (filesize % PACKAGE_SIZE != 0)
    app->totalPackages++;

  // allocate array to store file data
  unsigned char *filebuf = (unsigned char *)malloc(filesize);
  fread(filebuf, sizeof(unsigned char), filesize, file);

  // send start control package
```

```
  sendControlPackage(app->fd, C_START, filesize, app->filename);

  int packageSize = PACKAGE_SIZE;

  off_t idx = 0;
  while (idx < filesize)
  {
    unsigned char *package = getBufPackage(filebuf, &idx, &packageSize, filesize);
    if (sendDataPackage(app, package, packageSize, filesize) == -1)
      break;

    printProgressBar(app);
  }
  printf("\n");

  // send end control package
  sendControlPackage(app->fd, C_END, filesize, app->filename);

  // free file buf auxiliary array
  free(filebuf);

  // close file
  if (fclose(file) == -1)
  {
    printf("[closefile] - could not close file!\n");
    return -1;
  }

  return 1;
}

unsigned char *getControlPackage(unsigned char c, off_t filesize, char *filename,
int *packageSize)
{
  *packageSize = 9 + strlen(filename);
  unsigned char *package = (unsigned char *)malloc(*packageSize);

  if (!(c == C_START || c == C_END))
  {
    printf("[getControlPackage] - invalid control package state!\n");
    return NULL;
  }

  // package control state
  package[0] = c;
```

```
  // package first argument - filesize
  package[1] = T1;
  package[2] = L1;
  package[3] = (filesize >> 24) & 0xFF; // 1st byte
  package[4] = (filesize >> 16) & 0xFF; // 2nd byte
  package[5] = (filesize >> 8) & 0xFF;  // 3rd byte
  package[6] = filesize & 0xFF;         // 4th byte

  // package second argument - filepath
  package[7] = T2;
  package[8] = strlen(filename);
  int i;
  for (i = 0; i < strlen(filename); i++)
    package[i + 9] = filename[i];

  return package;
}

int sendControlPackage(int fd, int c, off_t filesize, char *filename)
{
  if (!(c == C_START || c == C_END))
  {
    printf("[sendControlPackage] - invalid control package state!\n");
    return -1;
  }

  // create start package with arguments filesize and filepath
  int packageSize = 0;
  unsigned char *package = getControlPackage(c, filesize, filename, &packageSize);

  // send it via serial port
  if (llwrite(fd, package, packageSize) != -1)
  {
    if (c == C_START)
      printf("[sendControlPackage] - START package sent!\n");
    else if (c == C_END)
      printf("[sendControlPackage] - END package sent!\n");

    return 1;
  }
  else
  {
    printf("[sendControlPackage] - could not send control package!\n");
    return -1;
  }
}
```

```
unsigned char *getBufPackage(unsigned char *filebuf, off_t *idx, int *packageSize,
off_t filesize)
{
  if (*idx + *packageSize > filesize)
    *packageSize = filesize - *idx;

  unsigned char *package = (unsigned char *)malloc(*packageSize);
  memcpy(package, filebuf + *idx, *packageSize);
  *idx += *packageSize;

  return package;
}

unsigned char *getDataPackage(Application *app, unsigned char *buf, int
*packageSize, off_t filesize)
{
  unsigned char *dataPackage = (unsigned char *)malloc(*packageSize + 4);

  // construct dataFrame
  dataPackage[0] = DATAFRAME_C;
  dataPackage[1] = app->package % 255;
  dataPackage[2] = (int)filesize / 256; // filesize MSB
  dataPackage[3] = (int)filesize % 256; // filesize LSB

  int i = 0;
  for (; i < *packageSize; i++)
    dataPackage[4 + i] = buf[i];

  app->package++;
  *packageSize += 4;

  return dataPackage;
}

int sendDataPackage(Application *app, unsigned char *buf, int packageSize, off_t
filesize)
{
  unsigned char *dataPackage = getDataPackage(app, buf, &packageSize, filesize);

  if (llwrite(app->fd, dataPackage, packageSize) == -1)
  {
    if (DEBUG)
      printf("[sendDataPackage] - could not send data package number %d!\n\n",
app->package);
```

```
    return -1;
  }
  else
  {
    if (DEBUG)
      printf("[sendDataPackage] - data package number %d sent!\n\n", app-
>package);

    return 1;
  }
}

// ----------- END OF APPLICATION LAYER -----------------

// ---------------- DATA LINK LAYER --------------------

DataLink *initDataLinkLayer()
{
  // allocate data link struct
  dl = (DataLink *)malloc(sizeof(DataLink));

  // initialize data link struct
  dl->frame = 0;
  dl->timeout = 0;
  dl->nTries = 0;

  return dl;
}

void destroyDataLinkLayer()
{
  // free datalink layer
  free(dl);
}

unsigned char *readControlFrame(int fd, unsigned char c)
{
  unsigned char *control = (unsigned char *)malloc(5 * sizeof(unsigned char));
  unsigned char byte;

  State state = INITIAL;
  while (!received && !dl->timeout)
  {
    read(fd, &byte, 1);

    switch (state)
```

```
    {
    case INITIAL:
      if (byte == FLAG)
      {
        control[0] = FLAG;
        state = STATE_FLAG;
      }
      break;
    case STATE_FLAG:
      if (byte == A_TRANSMITTER)
      {
        control[1] = A_TRANSMITTER;
        state = STATE_A;
      }
      else if (byte == FLAG)
      {
        control[0] = FLAG;
        state = STATE_FLAG;
      }
      else
      {
        state = INITIAL;
      }
      break;
    case STATE_A:
      if (byte == c)
      {
        control[2] = c;
        state = STATE_C;
      }
      else if (byte == FLAG)
      {
        control[0] = FLAG;
        state = STATE_FLAG;
      }
      else
      {
        state = INITIAL;
      }
      break;
    case STATE_C:
      if (byte == (control[1] ^ control[2]))
      {
        control[3] = control[1] ^ control[2];
        state = STATE_BCC;
      }
```

```
      else if (byte == FLAG)
      {
        control[0] = FLAG;
        state = STATE_FLAG;
      }
      else
      {
        state = INITIAL;
      }
      break;
    case STATE_BCC:
      if (byte == FLAG)
      {
        received = TRUE;
        control[4] = FLAG;
      }
      else
        state = INITIAL;
      break;
    default:
      break;
    }
  }

  return control;
}

unsigned char readControlC(int fd)
{
  unsigned char control[5];
  unsigned char byte;

  State state = INITIAL;
  while (!received && !dl->timeout)
  {
    read(fd, &byte, 1);

    switch (state)
    {
    case INITIAL:
      if (byte == FLAG)
      {
        control[0] = FLAG;
        state = STATE_FLAG;
      }
      break;
```

```
    case STATE_FLAG:
      if (byte == A_TRANSMITTER)
      {
        control[1] = A_TRANSMITTER;
        state = STATE_A;
      }
      else if (byte == FLAG)
      {
        control[0] = FLAG;
        state = STATE_FLAG;
      }
      else
      {
        state = INITIAL;
      }
      break;
    case STATE_A:
      if (byte == C_RR0 || byte == C_RR1 || byte == C_REJ0 || byte == C_REJ1)
      {
        control[2] = byte;
        state = STATE_C;
      }
      else if (byte == FLAG)
      {
        control[0] = FLAG;
        state = STATE_FLAG;
      }
      else
      {
        state = INITIAL;
      }
      break;
    case STATE_C:
      if (byte == (control[1] ^ control[2]))
      {
        control[3] = control[1] ^ control[2];
        state = STATE_BCC;
      }
      else if (byte == FLAG)
      {
        control[0] = FLAG;
        state = STATE_FLAG;
      }
      else
      {
        state = INITIAL;
```

```
      }
      break;
    case STATE_BCC:
      if (byte == FLAG)
      {
        received = TRUE;
        control[4] = FLAG;
      }
      else
        state = INITIAL;
      break;
    default:
      break;
    }
  }

  if (received)
  {
    unsigned char *frameC = (unsigned char *)malloc(sizeof(unsigned char));
    *frameC = control[2];
    return *frameC;
  }
  else
  {
    return 0xFF;
  }
}

int llopen(int fd)
{
  unsigned char setup[5];

  if (tcgetattr(fd, &dl->oldtio) == -1)
  { /* save current port settings */
    perror("[llopen] - tcgetattr error!\n");
    exit(-1);
  }

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

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

```
/*
VTIME e VMIN devem ser alterados de forma a proteger com um temporizador a
leitura do(s) próximo(s) caracter(es)
*/
dl->newtio.c_cc[VTIME] = 1; /* inter-character timer unused */
dl->newtio.c_cc[VMIN] = 0;  /* blocking read until 1 char is received */

tcflush(fd, TCIOFLUSH);

if (tcsetattr(fd, TCSANOW, &dl->newtio) == -1)
{
  perror("[llopen] - tcsetattr error!\n");
  exit(-1);
}
printf("[llopen] - new termios structure set\n");

setup[0] = FLAG;
setup[1] = A_TRANSMITTER;
setup[2] = C_SET;
setup[3] = setup[1] ^ setup[2]; //BCC
setup[4] = FLAG;

received = FALSE;
dl->nTries = 0;

unsigned char *ua;
while (!received && dl->nTries < MAX_TRIES)
{
  // writes SET to serial port
  write(fd, setup, 5);
  printf("[llopen] - SET sent: ");
  printArr(setup, 5);

  alarm(TIMEOUT);

  // read ua confirmation frame
  ua = readControlFrame(fd, C_UA);

  alarm(0);

  dl->timeout = 0;
  dl->nTries++;
}

if (received)
{
```

```
    printf("[llopen] - UA received: ");
    printArr(ua, 5);
    printf("[llopen] - connection opened successfully!\n");
    return TRUE;
  }
  else
  {
    return FALSE;
  }
}

int llclose(int fd)
{
  unsigned char disc[5], ua[5];

  disc[0] = FLAG;
  disc[1] = A_TRANSMITTER;
  disc[2] = C_DISC;
  disc[3] = disc[1] ^ disc[2]; //BCC
  disc[4] = FLAG;

  received = FALSE;
  dl->nTries = 0;

  while (!received && dl->nTries < MAX_TRIES)
  {
    /* writes SET to serial port */
    write(fd, disc, 5);
    printf("[llclose] - DISC sent: ");
    printArr(disc, 5);

    alarm(TIMEOUT);

    // read disc confirmation frame
    readControlFrame(fd, C_DISC);

    alarm(0);

    dl->timeout = 0;
    dl->nTries++;
  }

  if (received)
  {
    printf("[llclose] - DISC received: ");
    printArr(disc, 5);
```

```
    ua[0] = FLAG;
    ua[1] = A_TRANSMITTER;
    ua[2] = C_UA;
    ua[3] = ua[1] ^ ua[2];
    ua[4] = FLAG;

    write(fd, ua, 5);
    printf("[llclose] - UA sent: ");
    printArr(ua, 5);

    printf("[llclose] - connection successfully closed!\n");
  }

  if (tcsetattr(fd, TCSANOW, &dl->oldtio) == -1)
  {
    perror("[llclose] - tcsetattr error!\n");
    exit(-1);
  }

  if (close(fd) == -1)
  {
    printf("[llclose] - failed to close serial port!\n");
    return FALSE;
  }
  else
  {
    printf("[llclose] - serial port successfully closed!\n");
    return TRUE;
  }
}

void stuff(unsigned char *buf, int *bufsize)
{
  unsigned char *stuffedBuf = (unsigned char *)malloc(*bufsize);

  int i = 0, j = 0;
  for (; j < *bufsize; i++, j++)
  {
    if (buf[i] == FLAG)
    {
      stuffedBuf = (unsigned char *)realloc(stuffedBuf, ++(*bufsize));
      stuffedBuf[j] = ESCAPE;
      stuffedBuf[j + 1] = FLAG ^ 0x20;
      j++;
    }
```

```
    else if (buf[i] == ESCAPE)
    {
      stuffedBuf = (unsigned char *)realloc(stuffedBuf, ++(*bufsize));
      stuffedBuf[j] = ESCAPE;
      stuffedBuf[j + 1] = ESCAPE ^ 0x20;
      j++;
    }
    else
    {
      stuffedBuf[j] = buf[i];
    }
  }

  memcpy(buf, stuffedBuf, *bufsize);
  free(stuffedBuf);
}

int llwrite(int fd, unsigned char *buf, int bufSize)
{
  // allocate frame memory space
  int totalSize = bufSize + 6;
  unsigned char *frame = (unsigned char *)malloc(totalSize);

  // frame construction
  frame[0] = FLAG;
  frame[1] = A_TRANSMITTER;
  dl->frame == 0 ? (frame[2] = C_0) : (frame[2] = C_1);
  frame[3] = frame[1] ^ frame[2];

  // save bcc2 before stuffing
  int bcc2Size = 1;
  unsigned char *bcc2 = (unsigned char *)malloc(sizeof(unsigned char));
  bcc2[0] = buf[0];

  int i = 1;
  for (; i < bufSize; i++)
    bcc2[0] ^= buf[i];

  // data stuffing
  stuff(buf, &bufSize);
  if (bufSize != totalSize - 6)
  {
    // new totalsize, after stuffing data
    totalSize = bufSize + 6;
  }
```

```
for (i = 0; i < bufSize; i++)
  frame[4 + i] = buf[i];

// bcc2 stuffing
stuff(bcc2, &bcc2Size);
if (bcc2Size > 1)
{
  // in case bcc2 was stuffed
  frame = (unsigned char *)realloc(frame, totalSize + bcc2Size);

  int j = 0;
  for (; j < bcc2Size; j++)
    frame[4 + bufSize + j] = bcc2[j];
}
else
{
  frame[4 + bufSize] = bcc2[0];
}

frame[4 + bufSize + bcc2Size] = FLAG;

// send frame and wait confirmation
received = FALSE;
dl->nTries = 0;

while (!received && dl->nTries < MAX_TRIES)
{
  // eventually change BCC1 to simulate errors
  unsigned char *errorFrame = errorBCC(frame, totalSize);
  if (memcmp(frame, errorFrame, totalSize) != 0 && DEBUG)
    printf("\n[llwrite] - error inserted in BCC!\n");

  unsigned char *errorFrame2 = errorBCC2(errorFrame, totalSize);
  if (memcmp(errorFrame, errorFrame2, totalSize) != 0 && DEBUG) {
    printf("\n[llwrite] - error inserted in BCC2!\n");
  }

  // send frame
  write(fd, errorFrame2, totalSize);

  if (DEBUG)
  {
    printf("\n[llwrite] - frame sent: ");
    printArr(frame, totalSize);
  }
```

```
    alarm(TIMEOUT);

    // read control c parameter
    unsigned char c = readControlC(fd);
    if ((c == C_RR0 && dl->frame == 1) || (c == C_RR1 && dl->frame == 0))
    {
      received = TRUE;

      if (DEBUG)
        printf("[llwrite] - RR%x received for frame %d.\n", dl->frame ^ 1, dl-
>frame);

      dl->frame ^= 1;
    }
    else if (c == C_REJ0 || c == C_REJ1)
    {
      received = FALSE;
      dl->nTries = 0;

      if (DEBUG)
        printf("[llwrite] - REJ%x received for frame %d.\n", dl->frame ^ 1, dl-
>frame);
    }

    alarm(0);

    dl->nTries++;
    dl->timeout = 0;
  }

  if (received)
    return 1;
  else
    return -1;
}

// -------------- END OF DATA LINK LAYER ----------------

// --------------------- UTILS -------------------------

void timeoutHandler(int signo)
{
  if (signo == SIGALRM)
    dl->timeout = 1;
}
```

```
void installAlarm()
{
  struct sigaction action;
  action.sa_handler = timeoutHandler;
  sigemptyset(&action.sa_mask);
  action.sa_flags = 0;
  sigaction(SIGALRM, &action, NULL);
}

void uninstallAlarm()
{
  struct sigaction action;
  action.sa_handler = NULL;
  sigemptyset(&action.sa_mask);
  action.sa_flags = 0;
  sigaction(SIGALRM, &action, NULL);
}

unsigned char *errorBCC(unsigned char *package, int packageSize)
{
  unsigned char *errorPackage = (unsigned char *)malloc(packageSize);
  memcpy(errorPackage, package, packageSize);

  if ((rand() % 100) < BCC_PE)
  {
    // select a random character
    unsigned char randChar = (unsigned char)('A' + (rand() % 26));

    // insert character in A, C or BCC itself
    int errorIdx = (rand() % 3) + 1;
    errorPackage[errorIdx] = randChar;

    return errorPackage;
  }

  return package;
}

unsigned char *errorBCC2(unsigned char *package, int packageSize)
{
  unsigned char *errorPackage = (unsigned char *)malloc(packageSize);
  memcpy(errorPackage, package, packageSize);

  if ((rand() % 100) < BCC2_PE)
  {
    // select a random character
```

```
    unsigned char randChar = (unsigned char)('A' + (rand() % 26));

    // insert character in A, C or BCC itself
    int errorIdx = (rand() % (packageSize - 5)) + 4;
    errorPackage[errorIdx] = randChar;

    return errorPackage;
  }

  return package;
}

void printProgressBar(Application *app)
{
  float percentage = ((float)app->package / app->totalPackages) * 100;

  printf("\rProgress: [");

  int i = 0;
  for (; i < PROGRESS_BAR_DIM; i++)
  {
    if (i < percentage / (100 / PROGRESS_BAR_DIM))
      printf("#");
    else
      printf(" ");
  }

  printf("] %d%%", (int)percentage);
}

void printArr(unsigned char arr[], int size)
{
  int i;
  for (i = 0; i < size; i++)
    printf("%x ", arr[i]);
  printf("\n");
}

// ------------------ END OF UTILS ----------------------
```

reader.h

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

```c
#include <termios.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <strings.h>
#include <signal.h>

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

// ESCAPE flag
#define ESCAPE 0x7d

// data package flags
#define C_START 0x02 // control package start state byte
#define C_END 0x03   // control package end state byte

// information frame flags
#define PACKAGE_SIZE 100
#define C_0 0x00
#define C_1 0x40

// control frame flags
#define FLAG 0x7E
#define A_TRANSMITTER 0x03
#define A_RECEIVER 0x01
#define C_SET 0x03
#define C_DISC 0x0B
#define C_UA 0x07
#define C_RR0 0x05
#define C_RR1 0x85
#define C_REJ0 0x01
#define C_REJ1 0x81

// progress bar default dimension
#define PROGRESS_BAR_DIM 20

typedef enum
{
    INITIAL,
    STATE_FLAG,
    STATE_A,
    STATE_C,
```

```
    STATE_BCC,
    STATE_FINAL
} State;

typedef struct
{
    int fd;                 // serial port descriptor
    unsigned char *filename; // path of received file
    off_t filesize;         // filesize to be received
    int package;            // current package
    int totalPackages;      //totalpackages
} Application;

typedef struct
{
    int frame;                    // current frame number
    int expectedFrame;            // expected frame number
    struct termios oldtio, newtio; // structs storing terminal configurations
} DataLink;

// --------------- APPLICATION LAYER --------------------

Application *initApplicationLayer(char *port);

void destroyApplicationLayer(Application *app);

int receiveFile(Application *app);

unsigned char *parseStartPackage(Application *app);

int endPackageReceived(int fd, unsigned char *start, unsigned char *package, int
packageSize);

void parseDataPackage(unsigned char *package, int *packageSize);

// ----------- END OF APPLICATION LAYER -----------------

// ---------------- DATA LINK LAYER --------------------

DataLink *initDataLinkLayer();

void destroyDataLinkLayer();

int llopen(int fd);

int llclose(int fd);
```

```
unsigned char *readControlFrame(int fd, unsigned char c);

unsigned char *llread(int fd, int *frameSize);

void acceptFrame(int fd);

void rejectFrame(int fd);

// -------------- END OF DATA LINK LAYER ----------------

// --------------------- UTILS --------------------------

int receivedBcc2(unsigned char *frameI, int frameSize);

int saveFile(unsigned char *filepath, off_t filesize, unsigned char *filebuf);

void printProgressBar(Application *app);

void printArr(unsigned char arr[], int size);

// ------------------ END OF UTILS ----------------------
```

reader.c

```
#include "reader.h"
int DEBUG = FALSE;
int received = FALSE;
int success = FALSE;

DataLink *dl;

int main(int argc, char **argv)
{
  if ((argc < 2) ||
      ((strcmp("/dev/ttyS0", argv[1]) != 0) &&
       (strcmp("/dev/ttyS1", argv[1]) != 0)))
  {
    printf("Usage:\tnserial SerialPort\n\tex: nserial /dev/ttyS1\n");
    exit(1);
  }

  // initialize application layer
  Application *app = initApplicationLayer(argv[1]);
```

```
  // open connection
  if (!llopen(app->fd))
  {
    printf("llopen: failed to open connection!\n");
    return -1;
  }

  // receives file being transmitted
  receiveFile(app);

  // close connection
  if (!llclose(app->fd))
  {
    printf("llclose: failed to close connection!\n");
    return -1;
  }

  // destroy application layer
  destroyApplicationLayer(app);

  return 0;
}

// --------------- APPLICATION LAYER --------------------

Application *initApplicationLayer(char *port)
{
  Application *app = (Application *)malloc(sizeof(Application));

  /*
  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.
  */
  app->fd = open(port, O_RDWR | O_NOCTTY);
  if (app->fd < 0)
  {
    perror(port);
    exit(-1);
  }

  // initialize package numeration
  app->package = 0;

  // initialize datalink layer
  dl = initDataLinkLayer();
```

```
  return app;
}

void destroyApplicationLayer(Application *app)
{
  destroyDataLinkLayer();

  // free application layer
  free(app);
}

int receiveFile(Application *app)
{
  // receive first control package
  unsigned char *startPackage = parseStartPackage(app);
  if (startPackage != NULL)
    printf("receiveFile: START package received!\n");
  else
    return -1;

  // allocate array to store received file data
  unsigned char *filebuf = (unsigned char *)malloc(app->filesize);

  off_t idx = 0;
  unsigned char *package;

  while (TRUE)
  {
    int packageSize = 0;
    package = llread(app->fd, &packageSize);
    if (package == NULL)
      continue;

    // if end package is received
    if (endPackageReceived(app->fd, startPackage, package, packageSize))
      break;

    parseDataPackage(package, &packageSize);
    memcpy(filebuf + idx, package, packageSize);
    idx += packageSize;

    app->package++;
    printProgressBar(app);
  }

  saveFile(app->filename, app->filesize, filebuf);
```

```
  // deallocate buffer storing file data
  free(filebuf);

  return 1;
}

unsigned char *parseStartPackage(Application *app)
{
  int packageSize = 0;
  unsigned char *startPackage = llread(app->fd, &packageSize);

  if (startPackage == NULL || startPackage[0] != C_START)
  {
    printf("parseStartPackage: package is not START package!\n");
    return NULL;
  }

  // save file size
  off_t filesizeLength = startPackage[2];

  int i = 0;
  app->filesize = 0;
  for (; i < filesizeLength; i++)
    app->filesize |= (startPackage[3 + i] << (8 * filesizeLength - 8 * (i + 1)));

  // save file path
  int filenameLength = (int)startPackage[8];
  app->filename = (unsigned char *)malloc(filenameLength + 1);

  // building the c-string for filepath
  int j = 0;
  for (; j < filenameLength; j++)
    app->filename[j] = startPackage[9 + j];
  app->filename[j] = '\0';

  app->totalPackages = app->filesize / PACKAGE_SIZE;
  if (app->filesize % PACKAGE_SIZE != 0)
    app->totalPackages++;

  return startPackage;
}

int endPackageReceived(int fd, unsigned char *start, unsigned char *package, int
packageSize)
{
```

```
  // if received package is equal to the start package
  if (package[0] == C_END && !memcmp(start + 1, package + 1, packageSize - 1))
  {
    printf("\nreceiveFile: END package received!\n");
    return TRUE;
  }
  else
    return FALSE;
}

void parseDataPackage(unsigned char *package, int *packageSize)
{
  *packageSize -= 4;
  unsigned char *data = (unsigned char *)malloc(*packageSize);

  int i = 0;
  for (; i < *packageSize; i++)
    data[i] = package[4 + i];

  package = (unsigned char *)realloc(package, *packageSize);
  memcpy(package, data, *packageSize);

  free(data);
}

// ----------- END OF APPLICATION LAYER -----------------

// ---------------- DATA LINK LAYER ---------------------

DataLink *initDataLinkLayer()
{
  // allocate data link struct
  dl = (DataLink *)malloc(sizeof(DataLink));

  // initialize data link struct
  dl->frame = 0;
  dl->expectedFrame = 0;

  return dl;
}

void destroyDataLinkLayer()
{
  // free datalink layer
  free(dl);
}
```

```
int llopen(int fd)
{
  unsigned char ua[5];

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

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

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

  /*
  VTIME e VMIN devem ser alterados de forma a proteger com um temporizador a
  leitura do(s) próximo(s) caracter(es)
  */
  dl->newtio.c_cc[VTIME] = 0; /* inter-character timer unused */
  dl->newtio.c_cc[VMIN] = 1;  /* blocking read until 1 char is received */

  tcflush(fd, TCIOFLUSH);

  if (tcsetattr(fd, TCSANOW, &dl->newtio) == -1)
  {
    perror("llopen - tcsetattr error!");
    exit(-1);
  }
  printf("llopen - new termios structure set\n");

  // wait for SET control frame
  unsigned char *setup = readControlFrame(fd, C_SET);
  printf("llopen - SET received: ");
  printArr(setup, 5);

  // send UA confirmation
  ua[0] = setup[0];
  ua[1] = A_TRANSMITTER;
  ua[2] = C_UA;
  ua[3] = ua[1] ^ ua[2];
  ua[4] = setup[4];
```

```
  write(fd, ua, 5);
  printf("llopen - UA sent: ");
  printArr(ua, 5);

  printf("llopen - connection successfully opened!\n");
  return TRUE;
}

int llclose(int fd)
{
  // wait for DISC control frame
  unsigned char *disc = readControlFrame(fd, C_DISC);
  printf("llclose - DISC received: ");
  printArr(disc, 5);

  // send DISC confirmation frame
  write(fd, disc, 5);
  printf("llclose - DISC sent: ");
  printArr(disc, 5);

  // wait for UA control frame
  unsigned char *ua = readControlFrame(fd, C_UA);
  printf("llclose - UA received: ");
  printArr(ua, 5);

  tcsetattr(fd, TCSANOW, &dl->oldtio);

  printf("llclose - connection successfully closed!\n");

  return TRUE;
}

unsigned char *readControlFrame(int fd, unsigned char c)
{
  unsigned char *control = (unsigned char *)malloc(5 + sizeof(unsigned char));
  unsigned char byte;

  State state = INITIAL;
  received = FALSE;

  while (!received)
  {
    read(fd, &byte, 1);

    switch (state)
```

```
{
case INITIAL:
  if (byte == FLAG)
  {
    control[0] = FLAG;
    state = STATE_FLAG;
  }
  break;
case STATE_FLAG:
  if (byte == A_TRANSMITTER)
  {
    control[1] = A_TRANSMITTER;
    state = STATE_A;
  }
  else if (byte == FLAG)
  {
    control[0] = FLAG;
    state = STATE_FLAG;
  }
  else
  {
    state = INITIAL;
  }
  break;
case STATE_A:
  if (byte == c)
  {
    control[2] = c;
    state = STATE_C;
  }
  else if (byte == FLAG)
  {
    control[0] = FLAG;
    state = STATE_FLAG;
  }
  else
  {
    state = INITIAL;
  }
  break;
case STATE_C:
  if (byte == (control[1] ^ control[2]))
  {
    control[3] = control[1] ^ control[2];
    state = STATE_BCC;
  }
```

```
      else if (byte == FLAG)
      {
        control[0] = FLAG;
        state = STATE_FLAG;
      }
      else
      {
        state = INITIAL;
      }
      break;
    case STATE_BCC:
      if (byte == FLAG)
      {
        received = TRUE;
        control[4] = FLAG;
      }
      else
        state = INITIAL;
      break;
    default:
      break;
    }
  }

  return control;
}

unsigned char *llread(int fd, int *frameSize)
{
  int stuffedByte = 0;
  unsigned char control[5], byte, c;

  unsigned char *frameI = (unsigned char *)malloc(*frameSize);

  State state = INITIAL;
  received = FALSE, success = FALSE;

  while (!received)
  {
    read(fd, &byte, 1);

    switch (state)
    {
    case INITIAL:
      if (byte == FLAG)
        state = STATE_FLAG;
```

```
      break;
    case STATE_FLAG:
      if (byte == A_TRANSMITTER)
        state = STATE_A;
      else if (byte == FLAG)
        state = STATE_FLAG;
      else
        state = INITIAL;
      break;
    case STATE_A:
      if (byte == C_0)
      {
        c = byte;
        dl->frame = 0;
        state = STATE_C;
      }
      else if (byte == C_1)
      {
        c = byte;
        dl->frame = 1;
        state = STATE_C;
      }
      else if (byte == FLAG)
        state = STATE_FLAG;
      else
        state = INITIAL;
      break;
    case STATE_C:
      if (byte == (A_TRANSMITTER ^ c))
        state = STATE_BCC;
      else if (byte == FLAG)
        state = STATE_FLAG;
      else
        state = INITIAL;
      break;
    case STATE_BCC:
      // if FLAG is received
      if (byte == FLAG)
      {
        // if BCC2 was correct
        if (receivedBcc2(frameI, *frameSize))
          acceptFrame(fd);
        else
          rejectFrame(fd);

        received = TRUE;
```

```
      }
      // if ESCAPE is received
      else if (byte == ESCAPE)
        stuffedByte = 1;
      else if (stuffedByte)
      {
        frameI = (unsigned char *)realloc(frameI, ++(*frameSize));
        frameI[*frameSize - 1] = byte ^ 0x20;
        stuffedByte = 0;
      }
      else
      {
        frameI = (unsigned char *)realloc(frameI, ++(*frameSize));
        frameI[*frameSize - 1] = byte;
      }
      break;
    default:
      break;
    }
  }

  // remove bcc2 from data frame
  frameI = (unsigned char *)realloc(frameI, --(*frameSize));

  if (success)
  {
    return frameI;
  }
  else
  {
    free(frameI);
    return NULL;
  }
}

void acceptFrame(int fd)
{
  unsigned char control[5];

  if (dl->frame == 0)
  {
    // send C_RR1
    control[0] = FLAG;
    control[1] = A_TRANSMITTER;
    control[2] = C_RR1;
    control[3] = control[1] ^ control[2];
```

```
    control[4] = FLAG;

    write(fd, control, 5);
  }
  else if (dl->frame == 1)
  {
    // send C_RR0
    control[0] = FLAG;
    control[1] = A_TRANSMITTER;
    control[2] = C_RR0;
    control[3] = control[1] ^ control[2];
    control[4] = FLAG;

    write(fd, control, 5);
  }

  // if RR was sent
  if (dl->frame == dl->expectedFrame)
  {
    dl->expectedFrame ^= 1;
    success = TRUE;

    if (DEBUG)
      printf("\nllread: RR%x sent for frame %d!\n", dl->frame ^ 1, dl->frame);
  }
}

void rejectFrame(int fd)
{
  unsigned char control[5];

  if (dl->frame == 0)
  {
    // send C_REJ1
    control[0] = FLAG;
    control[1] = A_TRANSMITTER;
    control[2] = C_REJ1;
    control[3] = control[1] ^ control[2];
    control[4] = FLAG;

    write(fd, control, 5);
  }
  else if (dl->frame == 1)
  {
    // send C_REJ0
    control[0] = FLAG;
```

```
    control[1] = A_TRANSMITTER;
    control[2] = C_REJ0;
    control[3] = control[1] ^ control[2];
    control[4] = FLAG;

    write(fd, control, 5);
  }

  success = FALSE;

  if (DEBUG)
    printf("\nllread: REJ%x sent for frame %d!\n", dl->frame ^ 1, dl->frame);
}

// -------------- END OF DATA LINK LAYER ----------------

// -------------------- UTILS -------------------------

int receivedBcc2(unsigned char *frameI, int frameSize)
{
  if (frameSize == 0)
    return FALSE;

  unsigned char bcc2 = frameI[0];

  int i = 1;
  for (; i < frameSize - 1; i++)
    bcc2 ^= frameI[i];

  if (frameI[frameSize - 1] == bcc2)
    return TRUE;
  else
    return FALSE;
}

int saveFile(unsigned char *filename, off_t filesize, unsigned char *filebuf)
{
  // open new file
  FILE *file = fopen(filename, "wb");
  if (file == NULL)
  {
    perror("saveFile: could not open new file!\n");
    exit(-1);
  }

  if (fwrite(filebuf, sizeof(unsigned char), filesize, file) == -1)
```

```
  {
    printf("saveFile: error writing to file %s!\n", filename);
    return -1;
  }

  if (fclose(file) == -1)
  {
    printf("saveFile: error closing file %s!\n", filename);
    return -1;
  }

  return 1;
}

void printProgressBar(Application *app)
{
  float percentage = ((float)app->package / app->totalPackages) * 100;

  printf("\rProgress: [");

  int i = 0;
  for (; i < PROGRESS_BAR_DIM; i++)
  {
    if (i < percentage / (100 / PROGRESS_BAR_DIM))
      printf("#");
    else
      printf(" ");
  }

  printf("] %d%%", (int)percentage);
}

void printArr(unsigned char arr[], int size)
{
  int i;
  for (i = 0; i < size; i++)
    printf("%x ", arr[i]);
  printf("\n");
}

// ------------------ END OF UTILS ----------------------
```

Anexo II – dados para estudo da eficiência

Eficiência em função do tamanho do pacote

Tamanho do ficheiro = 10968 bytes = 87744 bits

C – Capacidade do canal de transmissão (bits/s) = 38400

Tamanho do pacote (bytes)	Tempo de transferência (s)	R – data rate (bits/s)	S - eficiência (R/C)
50	13.27	6612.21	0.1722
100	3.58	24509.50	0.6383
150	3.33	26349.55	0.6862
200	6.25	14039.04	0.3656
250	3.18	27592.45	0.7186
300	3.14	27943.95	0.7277

Eficiência em função da capacidade da ligação

Tamanho do ficheiro = 10968 bytes = 87744 bits

Tamanho do pacote (bytes) = 100

C - Capacidade da ligação (bits/s)	Tempo de transferência (s)	R – data rate (bits/s)	S - eficiência (R/C)
2400	56.80	1544.79	0.6437
4800	28.41	3088.49	0.6434
9600	14.22	6170.46	0.6428
19200	7.13	12306.31	0.6410
38400	3.58	24509.50	0.6383

Eficiência em função de erros no BCC1

Tamanho do ficheiro = 10968 bytes = 87744 bits

Tamanho do pacote (bytes) = 100

C – Capacidade do canal de transmissão (bits/s) = 38400

Probabilidade de erros no BCC1 (%)	Tempo de transferência (s)	R – data rate (bits/s)	S - eficiência (R/C)
0	3.58	24509.50	0.6383
2	9.58	9159.08	0.2385
4	15.59	5628.22	0.1466
8	21.59	4064.10	0.1058
10	27.58	3181.44	0.0829
20	72.50	1210.26	0.0315

Eficiência em função de erros no BCC2

Tamanho do ficheiro = 10968 bytes = 87744 bits

Tamanho do pacote (bytes) = 100

C – Capacidade do canal de transmissão (bits/s) = 38400

Probabilidade de erros no BCC2 (%)	Tempo de transferência (s)	R – data rate (bits/s)	S - eficiência (R/C)
0	3.58	24509.50	0.6383
4	3.68	23843.48	0.6209
8	3.83	22909.66	0.5966
10	3.91	22440.92	0.5844
20	4.48	19585.71	0.5100
40	6.27	13994.26	0.3644