Licenciatura em Engenharia Informática e Computação

Redes de Computadores

3LEIC025 - Turma 4 Grupo 9

João Pedro Rodrigues Coutinho up202108787

Miguel Jorge Medeiros Garrido up202108889

Sumário

Este trabalho realizado, proposto pela unidade curricular Redes de Computadores, tem como objetivo a implementação de um protocolo de comunicação de dados para a transmissão de ficheiros através da Porta Série RS-232.

Para a implementação deste protocolo, foi utilizada uma estratégia de *Stop-and-Wait* lecionada nas aulas teóricas.

Introdução

Baseado num guião previamente disponibilizado, este trabalho consistiu no desenvolvimento de um protocolo para transferir um ficheiro através de uma porta série. Este relatório tem como objetivo expor a implementação do protocolo anteriormente referido e está dividido em oito secções diferentes:

1. **Arquitetura** - Blocos funcionais e interfaces
2. **Estrutura do código** - APIs, principais estruturas de dados, principais funções e a sua relação com a arquitetura.
3. **Casos de uso principais** - identificação; sequências de chamada de funções.
4. **Protocolo de ligação lógica** - identificação dos principais aspetos funcionais na camada da ligação de dados
5. **Protocolo de aplicação** - identificação dos principais aspetos funcionais na camada da aplicação.
6. **Validação** - descrição dos testes efetuados com apresentação.
7. **Eficiência do protocolo de ligação de dados** - caracterização estatística da eficiência do protocolo, efetuada recorrendo a medidas sobre o código desenvolvido.
8. **Conclusão** - síntese da informação apresentada nas secções anteriores; reflexão sobre os objetivos de aprendizagem alcançados.

Arquitetura

Blocos Funcionais

Este projeto baseia-se na conjugação de duas camadas diferentes: ***LinkLayer*** e ***ApplicationLayer***.

A camada de ligação de dados, *LinkLayer*, encontra-se nos ficheiros *link_layer.h* e *link_layer.c* e tem como objetivo o estabelecimento de uma ligação entre dois computadores e o término da mesma, a criação e envio de tramas, a validação das tramas recebidas e a troca de mensagens sobre eventos bem sucedidos ou erros durante a conexão, transmissão ou desconexão.

A camada de aplicação, *ApplicationLayer*, encontra-se nos ficheiros *application_layer.h* e *application_layer.c*, e consiste na utilização da API da *LinkLayer* na receção e transmissão tanto de pacotes de controlo como de pacotes de dados. Nesta camada é possível definir o tamanho máximo desses pacotes, o número máximo de retransmissões e a velocidade de transferência de dados.

Interfaces

Para a execução deste programa são necessários dois computadores ligados entre si através de uma porta série ou da emulação desta em somente um computador utilizando três terminais diferentes - um emula o cabo que liga ambos os computadores, enquanto os os restantes dois executam como transmissor e recetor.

Estrutura do código

Na camada *LinkLayer*, foram utilizadas duas estruturas de dados fornecidas previamente: ***LinkLayer***, onde são definidos os parâmetros relacionados com a transferência de dados, e ***LinkLayerRole***, que tem como objetivo a identificação da função de cada computador - transmissor ou recetor. Para além destas estruturas, é criada uma adicional que facilita a leitura e interpretação da máquina de estados, ***message_state***.

```
typedef enum{
   LlTx,
   LlRx,
} LinkLayerRole;
```

```
typedef struct{
   char serialPort[50];
   LinkLayerRole role;
   int baudRate;
   int nRetransmissions;
   int timeout;
} LinkLayer;
```

```
enum message_state {
   START, FLAG_RCV, A_RCV, C_RCV, BCC_OK, DATA_READ, BCC_OK_2, ESC, END
};
```

O código seguinte representa as funções implementadas:

—----

```
// Criação de um alarme
void alarmHandler(int  signal)

// Estabelecimento da serial port
void establishSerialPort(LinkLayer connectionParameters)

// Reset da serial port
void resetPortSettings()

// Estabelecimento da ligação entre o transmissor e recetor
int llopen(LinkLayer connectionParameters)
int llSetFrame()
void llUaFrame()

// Aplicação de stuffing a um packet
void stuffing(unsigned char* frame, unsigned int packet_location, unsigned char
special, unsigned int *size)

// Envio de tramas
int llwrite(const unsigned char *buf, int bufSize)

// Leitura de tramas
int llread(unsigned char *packet)

// Encerramento da ligação entre o transmitter e o receiver
```

```
int  llclose(int  showStatistics)
int llcloseTx()
void llcloseRx()

// Estatísticas do Programa
void printStatistics()
```

Por outro lado, a utilização de estruturas de dados não foi necessária para o desenvolvimento da camada *ApplicationLayer*. Estas são as funções implementadas:

```
// Cria um pacote de dados
int buildControlPacket(int controlfield, const char * filename, int length)

// Interpreta um pacote de dados
int readControlPacket(unsigned char * name)

// Função de controlo de todas as outras
void applicationLayer(const  char *serialPort, const char *role, int baudRate, int nTries,
int timeout, const char *filename)
```

Casos de uso principais

A sequência de chamada do papel depende da função que cada computador executa, ainda que existam chamadas de funções semelhantes em ambos os casos.

Transmissor

1. ***llopen()*** - estabelece a ligação entre o transmissor e o recetor através de *llSetFrame()* no transmissor
2. ***llSetFrame()*** - escreve uma trama de controlo para o recetor e lê a resposta enviada por este
3. ***buildcontrolpacket()*** - constrói um pacote de controlo
4. ***llwrite()*** - forma uma trama a partir do pacote passado como argumento, envia-a ao recetor e interpreta a resposta recebida
5. ***llclose()*** - termina a ligação, utilizando a função *llcloseTx()* no caso do transmissor
6. ***llcloseTx()*** - envia uma trama de controlo para finalizar a ligação e recebe uma resposta do recetor

Recetor

1. ***llopen()*** - estabelece a ligação entre o transmissor e o recetor através de *llUaFrame()* no recetor
2. ***llUaFrame()*** - recebe a trama de controlo enviada pelo transmissor e envia uma resposta
3. ***readControlPacket()*** - intepreta os dados enviados no pacote de controlo
4. ***llread()*** - valida as tramas enviadas pelo transmissor e responde com uma mensagem, consoante o resultado do processo de validação
5. ***llclose()*** - termina a ligação, utilizando a função *llcloseRx()* no caso do recetor
6. ***llcloseRx()*** - obtém uma trama de controlo para desconectar e transmite a resposta

Protocolo de ligação lógica

O protocolo de ligação lógica é onde ocorre a interação entre o transmissor e o recetor através da porta série.

Para inicializar a ligação entre os dois computadores utilizamos a função ***llopen***, na qual serão chamadas as funções *llSetFrame* e *llUaFrame* no transmissor e no recetor, respetivamente. Inicialmente, o transmissor envia uma trama de supervisão SET e aguarda por uma resposta do recetor; este irá responder com uma trama de supervisão UA, caso a trama SET recebida anteriormente esteja correta. Se a receção da trama UA for bem-sucedida, a ligação fica operacional; se isto não se verificar, realizar-se-ão novas tentativas, dentro de um número limitado e com um intervalo de tempo específico, de enviar novamente a trama SET.

Com a conexão estabelecida, a função ***llwrite***, invocada pelo transmissor, é chamada e realiza um processo de *byte stuffing* sobre o pacote recebido, de modo a impedir a existência de erros causados pela interpretação errada da *flag* da trama. Por fim, adiciona ao pacote um *header* e um *trailer* - um processo de *framing*. Após isto, a trama é enviada para o recetor e, tal como no *llopen*, espera por uma resposta deste. Novamente, no caso da resposta revelar a rejeição da trama ou a não-receção desta, novas tentativas de envio da trama serão efetuadas até ser excedido o limite estabelecido. Se isto acontecer, ocorrerá uma tentativa de desconexão através da função *llclose*.

Na função ***llread***, chamada pelo recetor, é onde ocorre a leitura desta trama e sua consequente validação. Para isto, é necessário que se chegue ao fim da máquina de estados, utilizando o BCC1 para a validação do *header* e o BCC2 para a validação dos dados da trama após o processo de *destuffing*.

Para terminar a ligação, a função ***llclose*** é chamada por ambos. No caso do transmissor, é utilizada a função ***llcloseTx***, que envia uma trama de supervisão DISC e tal, como no *llopen*, aguarda por uma resposta - desta vez, no entanto, é esperado que a resposta seja algo igual ao que foi enviado. Em caso de sucesso, responde com o envio de uma trama de supervisão UA e termina a ligação e o programa. Por outro lado, o recetor chama a função ***llcloseRx*** e espera pela trama DISC, seguindo-se, em caso de sucesso, o envio de uma trama DISC semelhante e, por fim, pela receção de uma trama UA.

Protocolo de aplicação

Como referido anteriormente, é nesta camada que existe uma interação mais próxima com o utilizador. Através desta, são definidos os parâmetros cruciais para a execução do programa, tais como a porta série, a velocidade de transferência, o número máximo de bytes em cada pacote, o número de retransmissões, o tempo definido para cada alarme e o ficheiro a ser transferido.

Tendo a ligação sido confirmada, é gerado um pacote de controlo com o formato TLV no transmissor pela função ***buildControlPacket***, contendo informações acerca do tamanho do ficheiro e o nome deste. Do lado do recetor, este pacote é lido através da função ***readControlPacket***, sendo posteriormente criado um ficheiro cujo nome é o nome enviado no pacote concatenado com '-receiver'.

Com ajuda da API da *LinkLayer*, mais propriamente da função *llwrite*, vão ser enviados pacotes de dados, de tamanho previamente definido. A receção e interpretação destes é feita pelo recetor até que um dos pacotes recebidos possua o header que identifica um pacote de controlo, significando que o ficheiro já foi lido na sua totalidade. Outro cenário que interromperia a leitura e escrita no novo ficheiro seria uma falha no envio após 3 transmissões e consequente chamada da função *llclose* por parte do transmissor.

No final, quer a transmissão do ficheiro tenha tido êxito ou não, ambos invocam a função *llclose*, finalizando assim a ligação.

Validação

Vários testes foram efetuados ao longo do desenvolvimento do projeto, com o objetivo de validar a nossa implementação do protocolo de ligação de dados:

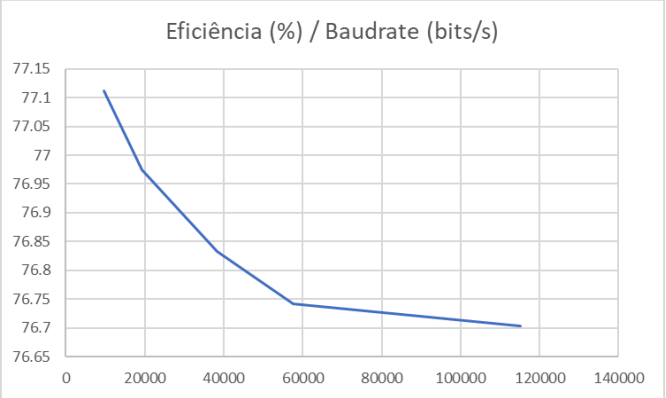
- Envio de ficheiros de diferentes tamanhos e nomes
- Transferência de ficheiros com diversos valores para o *baudrate*
- Transmissão de pacotes de dados com diferentes tamanhos
- Interrupção momentânea e completa da porta série
- Introdução de ruído através de um curto circuito na porta série

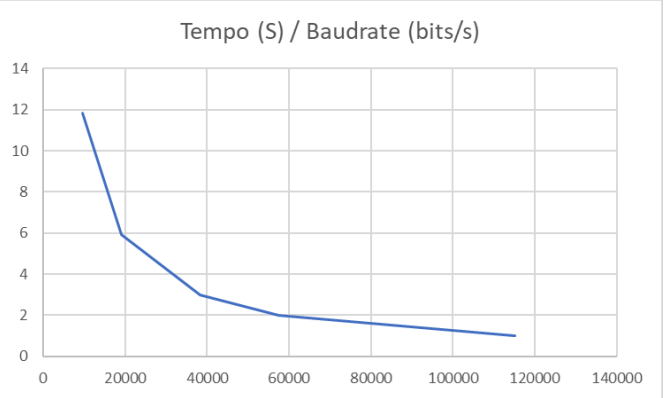
Todos os testes efetuados registaram os resultados pretendidos, tanto em ambiente controlado como na apresentação realizada no laboratório.

Eficiência do protocolo de ligação de dados

Baudrate

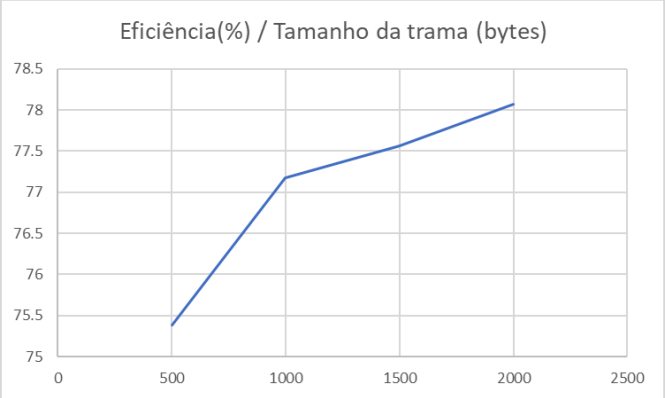
Foi observado, através da utilização de um ficheiro com tamanho fixo de 10968 bytes e número máximo de bytes em cada trama de 1000 bytes, que o tempo total da transferência e a velocidade de transmissão - o *baudrate* - são inversamente proporcionais; no entanto, a eficiência diminui com o aumento do *baudrate*.

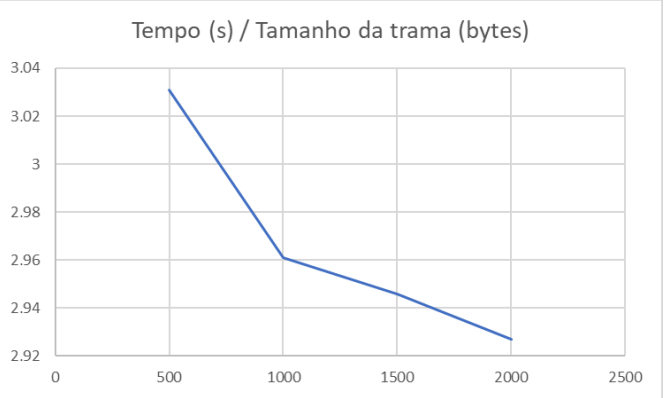




Tamanho da trama

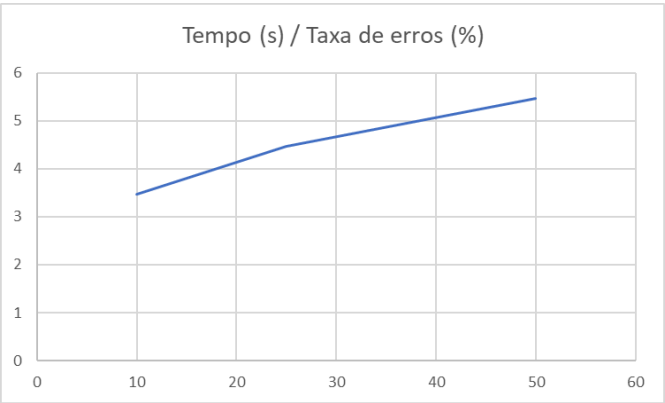
Mantendo o mesmo ficheiro e agora utilizando um *baudrate* fixo de 38400 bits/s, com a variação do número máximo de bytes em cada trama, verificou-se que o tempo total de transferência é novamente inversamente proporcional à variável em questão. No entanto, o aumento do tamanho de uma trama traduz-se numa maior eficiência.

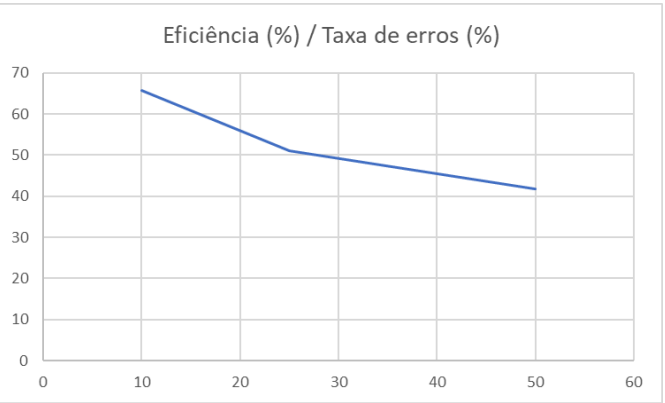




Taxa de erros

Utilizando todos os parâmetros anteriormente referidos com valores fixos - um ficheiro com 10968 bytes, um *baudrate* de 38400 bits/s e um tamanho da trama a 1500 bytes - e realizando alterações no *Frame Error Ratio* (FER), é possível verificar que um aumento deste resulta num aumento do tempo de transferência e, consequentemente, numa diminuição na eficiência do protocolo.





Conclusão

Concluindo, com a implementação de uma camada de ligação de dados, *LinkLayer*, e uma camada de aplicação, e através da sua comunicação entre si ,conseguimos atingir o objetivo proposto

Com a execução de trabalho conseguimos colocar em prática conceitos lecionados nas aulas teóricas tais como o mecanismo de *Stop-and-Wait* e *byte stuffing* que foram cruciais para a execução bem sucedida do projeto.

Anexo I - link_layer.h

```
// Link layer header.
// NOTE: This file must not be changed.

#ifndef _LINK_LAYER_H_
#define _LINK_LAYER_H_

typedef enum
{
    LlTx,
    LlRx,
} LinkLayerRole;

typedef struct
{
    char serialPort[50];
    LinkLayerRole role;
    int baudRate;
    int nRetransmissions;
    int timeout;
} LinkLayer;

// SIZE of maximum acceptable payload.
// Maximum number of bytes that application layer should send to link layer
#define MAX_PAYLOAD_SIZE 1000

// MISC
#define FALSE 0
#define TRUE 1

// Open a connection using the "port" parameters defined in struct linkLayer.
// Return "1" on success or "-1" on error.
int llopen(LinkLayer connectionParameters);

// Send data in buf with size bufSize.
// Return number of chars written, or "-1" on error.
int llwrite(const unsigned char *buf, int bufSize);

// Receive data in packet.
// Return number of chars read, or "-1" on error.
int llread(unsigned char *packet);

// Close previously opened connection.
// if showStatistics == TRUE, link layer should print statistics in the console on close.
// Return "1" on success or "-1" on error.
int llclose(int showStatistics);

#endif // _LINK_LAYER_H_
```

Anexo 2 - link_layer.c

```
// Link layer protocol implementation

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

#include "link_layer.h"

// MISC
#define _POSIX_SOURCE 1 // POSIX compliant source
#define BAUDRATE B38400

#define BUF_SIZE 5

int alarmEnabled = FALSE;
int alarmCount = 0;
int fd;
int llreadDisc = 0;
struct termios oldtio;
LinkLayerRole role;
unsigned char buf[BUF_SIZE];
enum message_state {
    START,
    FLAG_RCV,
    A_RCV,
    C_RCV,
    BCC_OK,
    DATA_READ,
    BCC_OK_2,
    ESC,
    END
};
int retransmissions;
unsigned int trans_frame = 0;
unsigned int prev_frame = 1;
double baud;
struct timeval start;
struct timeval end;

// Alarm function handler
void alarmHandler(int signal){
   alarmCount++;
   alarmEnabled = FALSE;
}
```

```
void establishSerialPort(LinkLayer connectionParameters) {
    // Program usage: Uses either COM1 or COM2
    const char *serialPortName = connectionParameters.serialPort;

    // 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(serialPortName, O_RDWR | O_NOCTTY);

    retransmissions = connectionParameters.nRetransmissions;
    role = connectionParameters.role;
    baud = (double)connectionParameters.baudRate;

    if (fd < 0)
    {
        perror(serialPortName);
        exit(-1);
    }

    struct termios newtio;

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

    // Clear struct for new port settings
    memset(&newtio, 0, sizeof(newtio));

    newtio.c_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;
    newtio.c_iflag = IGNPAR;
    newtio.c_oflag = 0;

    // Set input mode (non-canonical, no echo,...)
    newtio.c_lflag = 0;
    newtio.c_cc[VTIME] = 0; // Inter-character timer unused

    if (role == LlTx) {
        newtio.c_cc[VMIN] = 0;
    }
    else {
        newtio.c_cc[VMIN] = 1;
    }

    // VTIME e VMIN should be changed in order to protect with a
    // timeout the reception of the following character(s)

    // Now clean the line and activate the settings for the port
    // tcflush() discards data written to the object referred to
    // by fd but not transmitted, or data received but not read,
    // depending on the value of queue_selector:
    //   TCIFLUSH - flushes data received but not read.
    tcflush(fd, TCIOFLUSH);

    // Set new port settings
```

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

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

void resetPortSettings() {
    // Restore the old port settings
    if (tcsetattr(fd, TCSANOW, &oldtio) == -1)
    {
        perror("tcsetattr");
        exit(-1);
    }

    close(fd);
}

int llSetFrame() {
   // Set alarm function handler
   (void)signal(SIGALRM, alarmHandler);

   buf[0] = 0x7E;
   buf[1] = 0x03;
   buf[2] = 0x03;
   buf[3] = buf[1]^buf[2];
   buf[4] = 0x7E;

   alarm(3);
   alarmEnabled = TRUE;
   int count = 0;
   unsigned char byte;
   enum message_state state = START;
   int alarmExceeded = FALSE;
   int connected = FALSE;
   while (!connected && !alarmExceeded){
     int bytes = write(fd, buf, BUF_SIZE);
     state = START;
     while (state != END) {
       int reception = read(fd, &byte, 1);
       switch(state) {
         case START:
           if (byte == 0x7E)
             state = FLAG_RCV;
           break;
         case FLAG_RCV:
           if (byte == 0x03)
             state = A_RCV;
           else if (byte != 0x7E)
             state = START;
           break;
         case A_RCV:
           if (byte == 0x07){
             state = C_RCV;
             }
```

```
            else if (byte == 0x7E)
              state = FLAG_RCV;
            else
              state = START;
            break;
          case C_RCV:
            if (byte == 0x01^0x07){
              state = BCC_OK;
              }
            else if (byte == 0x7E)
              state = FLAG_RCV;
            else
              state = START;
            break;
          case BCC_OK:
            if (byte == 0x7E) {
              state = END;
              connected = TRUE;
            }
            else
              state = START;
            break;
          }

        if (alarmEnabled == FALSE && state != END){
          if (alarmCount > retransmissions) {
            alarm(0);
            alarmCount = 0;
            state = END;
            alarmExceeded = TRUE;
            return -1;
          }
          else {
            bytes = write(fd, buf, BUF_SIZE);
            alarm(3);
            alarmEnabled = TRUE;
          }
        }
      }
    }
    alarm(0);
    alarmCount = 0;
    alarmEnabled = FALSE;
    return 0;
}

void llUaFrame() {
    memset(buf, 0, BUF_SIZE);
    enum message_state state = START;
    unsigned char byte;

    while (state != END) {
        int bytes = read(fd, &byte, 1);
        switch(state) {
            case START:
              if (byte == 0x7E){
                state = FLAG_RCV;
```

```
            buf[START] = byte;
            }
          break;
        case FLAG_RCV:
          if (byte == 0x03){
            state = A_RCV;
            buf[FLAG_RCV] = byte;}
          else if (byte != 0x7E)
            state = START;
          break;
        case A_RCV:
          if (byte == 0x03){
            state = C_RCV;
            buf[A_RCV] = 0x07;}
          else if (byte == 0x7E)
            state = FLAG_RCV;
          else
            state = START;
          break;
        case C_RCV:
          if (byte == 0x03^0x03){
            state = BCC_OK;
            buf[C_RCV] = buf[FLAG_RCV]^buf[A_RCV];}
          else if (byte == 0x7E)
            state = FLAG_RCV;
          else
            state = START;
          break;
        case BCC_OK:
          if (byte == 0x7E) {
            state = END;
            buf[BCC_OK] = byte;
          }
          else
            state = START;
          break;
        }
    }
    int bytes = write(fd, buf, BUF_SIZE);
}

////////////////////////////////////////////////
// LLOPEN
////////////////////////////////////////////////
int llopen(LinkLayer connectionParameters)
{
  establishSerialPort(connectionParameters);
  gettimeofday(&start, NULL);
  int connection;
  if (role == LlTx) {
    connection = llSetFrame();
  }
  else {
    llUaFrame();
  }
  return connection;
}
```

```
////////////////////////////////////////////////
// LLWRITE
////////////////////////////////////////////////
void stuffing(unsigned char* frame, unsigned int packet_location, unsigned char special,
unsigned int *size) {
  frame = realloc(frame, ++(*size));
  frame[packet_location] = 0x7D;
  frame[packet_location+1] = special^0x20;
}

int llwrite(const unsigned char *buf, int bufSize)
{
  unsigned int bufSizeParam = bufSize+6;
  unsigned char* frame = (unsigned char*) malloc (bufSizeParam);
  memset(frame, 0, bufSize+6);
  *frame = 0x7E;
  *(frame+1) = 0x03;
  if (trans_frame == 0) {
    *(frame+2) = 0x00;
  }
  else if (trans_frame == 1) {
    *(frame+2) = 0x40;
  }
  *(frame+3) = *(frame+1)^*(frame+2);

  unsigned char bcc_2;
  bcc_2 = *buf;
  for (unsigned int i = 1 ; i < bufSize ; i++) {
    bcc_2 ^= *(buf+i);
  }
  unsigned int packet_loc = 4;

  for (unsigned int i = 0 ; i < bufSize ; i++) {
    if (*(buf+i) == 0x7E) {
      stuffing(frame, packet_loc, 0x7E, &bufSizeParam);
      packet_loc++;
    }
    else if (*(buf+i) == 0x7D) {
      stuffing(frame, packet_loc, 0x7D, &bufSizeParam);
      packet_loc++;
    }
    else {
      *(frame+packet_loc) = *(buf+i);
    }
    packet_loc++;
  }

  if (bcc_2 == 0x7E) {
    stuffing(frame, packet_loc, 0x7E, &bufSizeParam);
    packet_loc+=2;
  }
  else if (bcc_2 == 0x7D) {
    stuffing(frame, packet_loc, 0x7D, &bufSizeParam);
    packet_loc+=2;
  }
  else {
```

```
  *(frame+packet_loc) = bcc_2;
  packet_loc++;
}
*(frame+packet_loc) = 0x7E;
packet_loc++;

alarm(3);
alarmEnabled = TRUE;
unsigned char byte;
unsigned char cbyte;
int accepted = FALSE;
int alarmExceeded = FALSE;
enum message_state state = START;
while (!accepted && !alarmExceeded){
  write(fd, frame, packet_loc);
  state = START;
  while (state != END) {
    read(fd, &byte, 1);
    switch (state) {
      case START:
          if (byte == 0x7E) {
            state = FLAG_RCV;
          }
          break;
      case FLAG_RCV:
          if (byte == 0x03) {
            state = A_RCV;
          }
          else if (byte == 0x7E) {
            state = FLAG_RCV;
          }
          else {
            state = START;
          }
          break;
      case A_RCV:
          if (byte == 0x05 || byte == 0x85){  // RR0, RR1
            accepted = TRUE;
            state = C_RCV;
            cbyte = byte;
          }
          else if(byte == 0x01 || byte == 0x81) {   //REJ0, REJ1
            state = C_RCV;
            cbyte = byte;
          }
          else if (byte == 0x7E) {
            state = FLAG_RCV;
          }
          else {
            state = START;
          }
          break;
      case C_RCV:
          if (byte == (0x03^cbyte)) {
            state = BCC_OK;
          }
          else if (byte == 0x7E) {
```

```
                state = FLAG_RCV;
              }
              else {
                state = START;
              }
              break;
          case BCC_OK:
              if (byte == 0x7E){
                state = END;
              }
              else {
                state = START;
              }
              break;
          default:
              break;
        }
        if (alarmEnabled == FALSE && state != END){
          if (alarmCount > retransmissions) {
            alarm(0);
            alarmExceeded = TRUE;
            state = END;
          }
          else{
            int bytes = write(fd, frame, packet_loc);
            alarm(3);
            alarmEnabled = TRUE;
          }
        }
      }
    }
    alarm(0);
    alarmCount = 0;
    alarmEnabled = FALSE;
    if (accepted) {
      trans_frame = 1 - trans_frame;
      return packet_loc;
    }
    else {
      return -1;
    }
}

////////////////////////////////////////////////
// LLREAD
////////////////////////////////////////////////
int llread(unsigned char *packet)
{
    unsigned char tmp[2050];
    enum message_state state = START;
    unsigned char byte;
    int bcc = 0;
    int size = 0;
    while (state != END) {
      int bytes = read(fd, &byte, 1);
      switch(state) {
        case START:
```

```
    if (byte == 0x7E){
      state = FLAG_RCV;
      }
    break;
  case FLAG_RCV:
    if (byte == 0x03){
      state = A_RCV;
      }
    else if (byte != 0x7E)
      state = START;
    break;
  case A_RCV:
    if (byte == 0x00){
      trans_frame = 0;
      state = C_RCV;
    }
    else if (byte == 0x40){
      trans_frame = 1;
      state = C_RCV;
    }
    else if (byte == 0x0B) {
      llreadDisc = 1;
      return -2;
    }
    else if (byte == 0x7E)
      state = FLAG_RCV;
    else
      state = START;
    break;
  case C_RCV:
    if (byte == 0x03^0x00 || byte == 0x03^0x40){
      state = BCC_OK;
      }
    else if (byte == 0x7E)
      state = FLAG_RCV;
    else
      state = START;
    break;
  case BCC_OK:
    if (byte == 0x7D)
      state = ESC;
    else if (byte == 0x7E){
      int bcc;
      size-=1;
      bcc = tmp[0];
      for (unsigned int i = 1 ; i < size; i++) {
        bcc ^= tmp[i];
      }
      if (bcc == tmp[size]){
        memcpy(packet,tmp,MAX_PAYLOAD_SIZE);
        state = END;
        buf[0]=0x7E;
        buf[1]=0x03;
        if (trans_frame)
          buf[2]=0x05;
        else
          buf[2]=0x85;
```

```
            buf[3]=buf[1]^buf[2];
            buf[4]=0x7E;
            write(fd,buf,BUF_SIZE);
            if (prev_frame != trans_frame){
              prev_frame = trans_frame;
              return size;
            }
            return -1;
            write(fd,buf,BUF_SIZE);
            return size;
          }
          else{
            buf[0]=0x7E;
            buf[1]=0x03;
            if (trans_frame)
               buf[2]=0x01;
            else
               buf[2]=0x81;
            buf[3]=buf[1]^buf[2];
            buf[4]=0x7E;
            write(fd,buf,BUF_SIZE);
            return -1;
          }

        }
        else {
          tmp[size] = byte;
          size++;
        }
        break;
      case ESC:
        state = BCC_OK;
        if (byte == 0x5E){
          tmp[size]=0x7E;
          size++;
        }
        else if (byte == 0x5D){
          tmp[size]=0x7D;
          size++;
        }
        break;
    }
  }
}

////////////////////////////////////////////////
// LLCLOSE
////////////////////////////////////////////////

/*llclose transmitter handler*/
int llcloseTx(){
  buf[0] = 0x7E;
  buf[1] = 0x03;
  buf[2] = 0x0B;
  buf[3] = buf[1]^buf[2];
  buf[4] = 0x7E;
```

```
alarm(3);
alarmEnabled=TRUE;
unsigned char byte;
enum message_state state = START;
int alarmExceeded = FALSE;
int disconnected = FALSE;
while (!disconnected && !alarmExceeded){
  int bytes = write(fd, buf, BUF_SIZE);
  state = START;
  while (state != END){
    int reception = read(fd, &byte, 1);
    switch(state) {
      case START:
        if (byte == 0x7E)
          state = FLAG_RCV;
        break;
      case FLAG_RCV:
        if (byte == 0x03)
          state = A_RCV;
        else if (byte != 0x7E)
          state = START;
        break;
      case A_RCV:
        if (byte == 0x0B)
          state = C_RCV;
        else if (byte == 0x7E)
          state = FLAG_RCV;
        else
          state = START;
        break;
      case C_RCV:
        if (byte == 0x01^0x0B)
          state = BCC_OK;
        else if (byte == 0x7E)
          state = FLAG_RCV;
        else
          state = START;
        break;
      case BCC_OK:
        if (byte == 0x7E) {
          state = END;
          disconnected = TRUE;
        }
        else
          state = START;
        break;
    }

    if (alarmEnabled == FALSE && state != END){
      if (alarmCount > retransmissions) {
        alarm(0);
        state = END;
        alarmExceeded = TRUE;
        return -1;
      }
      else{
        int bytes = write(fd, buf, BUF_SIZE);
```

```
            alarm(3);
            alarmEnabled = TRUE;
          }
        }
      }
    }
    alarm(0);
    alarmEnabled = FALSE;
    alarmCount = 0;
    buf[0] = 0x7E;
    buf[1] = 0x01;
    buf[2] = 0x07;
    buf[3] = buf[1]^buf[2];
    buf[4] = 0x7E;

    int bytes = write(fd, buf, BUF_SIZE);
    return 0;
}

/*llclose receiver handler*/
void llcloseRx(){
    enum message_state state = START;
    unsigned char byte;
    if (llreadDisc){
      state = C_RCV;
      }
    while (state != END) {
      int bytes = read(fd, &byte, 1);
      switch(state) {
        case START:
          if (byte == 0x7E)
            state = FLAG_RCV;
          break;
        case FLAG_RCV:
          if (byte == 0x03)
            state = A_RCV;
          else if (byte != 0x7E)
            state = START;
          break;
        case A_RCV:
          if (byte == 0x0B)
            state = C_RCV;
          else if (byte == 0x7E)
            state = FLAG_RCV;
          else
            state = START;
          break;
        case C_RCV:
          if (byte == 0x03^0x0B)
            state = BCC_OK;
          else if (byte == 0x7E)
            state = FLAG_RCV;
          else
            state = START;
          break;
        case BCC_OK:
          if (byte == 0x7E) {
```

```
            state = END;
          }
          else
            state = START;
          break;
        }
    }
    buf[0] = 0x7E;
    buf[1] = 0x03;
    buf[2] = 0x0B;
    buf[3] = buf[1]^buf[2];
    buf[4] = 0x7E;
    int bytes = write(fd, buf, BUF_SIZE);
    state = START;
    while (state != END) {
      int bytes = read(fd, &byte, 1);

      switch(state) {
        case START:
          if (byte == 0x7E)
            state = FLAG_RCV;
          break;
        case FLAG_RCV:
          if (byte == 0x01)
            state = A_RCV;
          else if (byte != 0x7E)
            state = START;
          break;
        case A_RCV:
          if (byte == 0x07)
            state = C_RCV;
          else if (byte == 0x7E)
            state = FLAG_RCV;
          else
            state = START;
          break;
        case C_RCV:
          if (byte == 0x03^0x07)
            state = BCC_OK;
          else if (byte == 0x7E)
            state = FLAG_RCV;
          else
            state = START;
          break;
        case BCC_OK:
          if (byte == 0x7E) {
            state = END;
          }
          else
            state = START;
          break;
        }
    }
}

/*Show program's statistics*/
void printStatistics() {
```

```
        double cpu_time = ((double)((end.tv_sec + end.tv_usec*0.000001) - (start.tv_sec +
start.tv_usec*0.000001)));
        double transfer_rate = (double)((10968*8) / cpu_time);
        double efficiency = transfer_rate / baud;
        printf("CPU Time Used: %f seconds\n", cpu_time);
        printf("Transfer Rate: %f bits/s\n", transfer_rate);
        printf("Efficiency: %f %%\n", efficiency);
        printf("Maximum Payload Size: %d\n", MAX_PAYLOAD_SIZE);
    }

    int llclose(int showStatistics){
      int connection;
        if (role == LlTx) {
            connection = llcloseTx();
        }
        else {
            llcloseRx();
        }

        gettimeofday(&end, NULL);
        if (showStatistics) {
            printStatistics();
        }

        resetPortSettings();

        return connection;
    }
```

Anexo 3 - application_layer.h

```
// Application layer protocol header.
// NOTE: This file must not be changed.

#ifndef _APPLICATION_LAYER_H_
#define _APPLICATION_LAYER_H_

// Application layer main function.
// Arguments:
//   serialPort: Serial port name (e.g., /dev/ttyS0).
//   role: Application role {"tx", "rx"}.
//   baudrate: Baudrate of the serial port.
//   nTries: Maximum number of frame retries.
//   timeout: Frame timeout.
//   filename: Name of the file to send / receive.
void applicationLayer(const char *serialPort, const char *role, int baudRate,
                      int nTries, int timeout, const char *filename);

#endif // _APPLICATION_LAYER_H_
```

Anexo 4 - application_layer.c

```
// Application layer protocol implementation

#include "application_layer.h"
#include "link_layer.h"

#include <string.h>
#include <stdio.h>
#include <stdlib.h>

//Creates a control packet
int buildControlPacket(int controlfield, const char* filename, int length){
    int lensize = 0;
    int tmp = length;

    while (tmp > 0) {
        tmp >>= 8;
        lensize++;
    }

    int namesize = strlen(filename);
    int size = 5+lensize+namesize;
    unsigned char control[size];
    int i = 0;

    control[i++] = controlfield;
    control[i++] = 0;
    control[i] = lensize;
    for (int j = i + lensize; j > i; j--){
        control[j] = length & 0xFF;
        length >>= 8;
    }
    i+=lensize+1;
    control[i++] = 1;
    control[i++] = namesize;

    memcpy(control+i,filename,namesize);
    return llwrite(control, size);
}

//Reads a control packet
int readControlPacket(unsigned char* name){
    unsigned char control[MAX_PAYLOAD_SIZE];
    int reada;
    while ((reada = llread(control)) == -1);
    if (reada == -2){
        return -2;
    }
```

```
            int size = 0;
            int filesize = control[2];
            int i;
            for (i = 3; i < 3+filesize; i++){
               size += control[i];
               if (i+1 < 3+filesize){
                  size <<= 8;
               }
            }

            int namesize = control[++i];
            memcpy(name,control+(++i), namesize);
            name[namesize]='\0';

            int format_pos = -1;
            for (int i = 0; i < namesize; i++) {
               if (name[i] == '.' && (i + 3) < namesize && name[i + 1] == 'g' && name[i + 2] == 'i'
&& name[i + 3] == 'f') {
                  format_pos = i;
                  break;
               }
            }

            if (format_pos != -1) {
               memmove(name + format_pos, "-received.gif", 14);
            }
            return 0;
         }

         void applicationLayer(const char *serialPort, const char *role, int baudRate, int nTries, int
timeout, const char *filename) {
            LinkLayer parameters;
            strcpy(parameters.serialPort, serialPort);
            if (strcmp(role, "rx") == 0) {
               parameters.role = LlRx;
            }
            else {
               parameters.role = LlTx;
            }
            parameters.baudRate = baudRate;
            parameters.nRetransmissions = nTries;
            parameters.timeout = timeout;

            int statistics = 0; /*change to 1 if statistics are pretended*/
            if (llopen(parameters) < 0){
               perror("Connection failed\n");
               exit(EXIT_FAILURE);
            }
            if (parameters.role == LlRx) {
               unsigned char name[MAX_PAYLOAD_SIZE];
               if (readControlPacket(name) == -2){
                  perror("Error transfering the control\n");
                  llclose(statistics);
                  exit(EXIT_FAILURE);
               }
               FILE *fptr = fopen(name, "wb+");
               int read;
```

```
    unsigned char data[MAX_PAYLOAD_SIZE];
    do {
        while ((read = llread(data)) == -1);
        if (read == -2){
            perror("Error transfering the data\n");
            fclose(fptr);
            llclose(statistics);
            exit(EXIT_FAILURE);
        }
        if (data[0] == 3) break;
        fwrite(data+3, 1, read-3, fptr);
        fflush(fptr);
    } while (1);

    fclose(fptr);
    llclose(statistics);
}
else if (parameters.role == LlTx) {
    FILE *fptr;
    unsigned int start_ctrl = 2;
    unsigned int end_ctrl = 3;

    fptr = fopen(filename, "rb");

    if (fptr == NULL) {
        perror("This file wasn't found\n");
        exit(EXIT_FAILURE);
    }

    fseek(fptr, 0, SEEK_END);
    int len = ftell(fptr);
    fseek(fptr, 0, SEEK_SET);

    if (buildControlPacket(start_ctrl, filename, len) == -1){
        perror("Control packet error\n");
        fclose(fptr);
        llclose(statistics);
        exit(EXIT_FAILURE);
    }

    int bytesleft = len;
    int datasize;
    unsigned char data[MAX_PAYLOAD_SIZE-3];
    unsigned char data_packet[MAX_PAYLOAD_SIZE];

    while (bytesleft > 0){
        data_packet[0] = 1;
        if (bytesleft > MAX_PAYLOAD_SIZE-3){
            datasize = MAX_PAYLOAD_SIZE-3;
            bytesleft -= datasize;
        }
        else{
            datasize = bytesleft;
            bytesleft -= datasize;
        }

        fread(data, 1, datasize, fptr);
```

```
            data_packet[1] = datasize >> 8 & 0xFF;
            data_packet[2] = datasize & 0xFF;
            memcpy(data_packet + 3, data, datasize);
            int written = llwrite(data_packet, datasize+3);
            if (written == -1)
                break;
        }

        if (buildControlPacket(end_ctrl, filename, len) == -1){
            perror("Control packet error\n");
            fclose(fptr);
            llclose(statistics);
            exit(EXIT_FAILURE);
        }

        fclose(fptr);

        if (llclose(statistics) == -1){
            perror("Error disconnecting\n");
            exit(EXIT_FAILURE);
        }
    }
    else {
        perror("Unidentified Role\n");
        exit(EXIT_FAILURE);
    }
}
```