1º Trabalho Laboratorial

Protocolo de Ligação de Dados

Ana Sofia Oliveira Teixeira	up201806629
Diogo Simão Correia Gomes	up201806629
Tiago André Carneiro Bessa	up201606796

Índice

1 Sumário

Este relatório foi elaborado de acordo com o trabalho laboratorial realizado no âmbito da unidade curricular de Redes de Computadores. Teve como objetivo a implementação de um protocolo de dados com uma aplicação de transferência de ficheiros entre dois computadores de forma assíncrona através de uma porta-série. A aplicação é capaz de transferir ficheiros corretamente e, no caso de ocorrência de erros, é capaz de recuperar os dados para realizar a transferência completamente.

2 Introdução

O objetivo do trabalho laboratorial era implementar um protocolo de dados, de acordo com as instruções fornecidas, através de uma aplicação de transferência de dados. No caso do relatório, este tem como objetivo detalhar a componente teórica do trabalho, explicitando a implementação utilizada e o seu funcionamento. Este terá a seguinte estrutura:

- **Arquitetura** - descrição dos blocos funcionais e interfaces implementadas;
- **Estrutura do código** - apresentação das APIs, principais estruturas de dados e funções e a sua relação com a arquitetura;
- **Casos de uso principais** - identificação dos principais casos de uso e das sequências de chamadas de funções;
- **Protocolo de ligação lógica** - identificação dos principais aspetos funcionais;
- **Protocolo de aplicação** - identificação dos principais aspetos funcionais;
- **Validação** - descrição dos testes efetuados;
- **Eficiência do protocolo de ligação de dados** - caracterização estatística da eficiência do protocolo;
- **Conclusões** - reflexão sobre os objetivos de aprendizagem alcançados.

3 Arquitetura

O trabalho está dividido em duas camadas: a camada de ligação de dados e a camada da aplicação. A primeira é responsável pelas interações com a porta-série, tais como a abertura, o fecho, a leitura e a escrita de dados e pelo tratamento de tramas (delineamento, stuffing, proteção e retransmissão). No caso da segunda, esta é responsável pelo envio e receção dos ficheiros, processamento do serviço (tratamento de cabeçalhos e distinção entre pacotes de controlo e de dados).

4 Estrutura do código

Para o desenvolvimento do trabalho laboratorial, foram utilizadas diferentes estruturas de dados e funções descritas de seguida:

4.1 program.c

- **main** - função partilhada pela camada de aplicação e pela camada de ligação.

4.2 physical_layer.h physical_layer.c

- **open_serial_port** - função que abre a porta-série;
- **close_serial_port** - função que fecha a porta-série.

4.3 datalink_layer.h datalink_layer.c

- **alarm_handler** - função que é chamada para verificar se o número de tentativas escolhido foi ultrapassado ou não;
- **state_machine** - função que inclui a máquina de estados que permite o mecanismo de espera;
- **SET_transmitter** - função que espera pelo sinal UA para poder prosseguir com a transmissão, no caso do transmissor;
- **SET_receiver** - função que lança o sinal UA no caso de receber a confirmação por parte do transmissor de que está pronto para a transferência de dados;
- **llopen** - função que abre a porta-série e troca as tramas SET;
- **llwrite** - função que envia as tramas I de acordo com a máquina de estados;
- **llread** - função que lê as tramas I e envia a trama de supervisão;
- **llclose** - função que faz a troca de tramas DISC;
- **create_frame** - função que cria as frames das tramas I;
- **frame_header** - função que cria os cabeçalhos das frames;
- **remove_supervision_frame** - função que remove a frame de supervisão para que possamos ficar apenas com os dados do ficheiro em si;
- **stuffing** - função que realiza byte stuffing para proteger as mensagens;
- **destuffing** - função que realiza destuffing para proteger as mensagens;
- **BCC2** - função que realiza o BCC2 para verificar que existem erros na mensagem;
- **BCC1_random_error** - função que informa que existiu um erro no BCC1;
- **BCC2_random_error** - função que informa que existiu um erro no BCC2;
- **send_RR_REJ** - função que envia a trama de RR ou de REJ. Envia RR no caso de ser enviado corretamente e REJ no caso oposto;
- **send_DISC** - função que envia a trama de DISC;

4.4 application_layer.h application_layer.c

- **send_message** - função que faz a ligação com a camada da ligação de dados para escrever as mensagens;
- **get_message** - função que faz a ligação com a camada da ligação de dados para receber as mensagens, tanto no caso de verificação, como de parâmetros do ficheiro ou os dados em si;
- **get_only_data** - função que apenas lê e guarda os dados do ficheiro a ser transferido;
- **data_package** - função que coloca informação nos tramas;
- **START_END_package** - função que inicia ou termina os tramas;
- **file_size** - função que retorna o tamanho do ficheiro a ser transferido;
- **file_parameters** - função que verifica os parâmetros dos ficheiros;
- **readfile** - função que lê os dados;
- **writefile** - função que escreve os dados;
- **verify** - função que verifica o tamanho do ficheiro;

5 Casos de uso principais

No caso do transmissor, este deve correr o código da seguinte forma: ./program /dev/ttyS0 w <file> <size> <timeout> <retransmissions> sendo:

- program o nome da aplicação;
- /dev/ttyS0 a porta;
- w a flag que indica que o programa está a correr como transmissor;
- <file> o caminho do ficheiro a ser enviado;

- <size> o tamanho da informação que cada trama envia;
- <timeout> o tempo máximo de espera até parar a transmissão;
- <retransmissions> o número máximo de tentativas de retransmissões até o programa desistir de tentar retransmitir os dados.

No caso do receptor, este deve correr o código da seguinte forma: ./program /dev/ttyS0 r <timeout> <retransmissions> sendo:

- program o nome da aplicação;
- /dev/ttyS0 a porta;
- r a flag que indica que o programa está a correr como receptor;
- <timeout> o tempo máximo de espera até parar a transmissão;
- <retransmissions> o número máximo de tentativas de retransmissões até o programa desistir de tentar retransmitir os dados.

A sequência de acontecimentos após a chamada do programa, no caso do transmissor, é a seguinte:

1. Abertura da ligação entre os computadores;
2. Envio da trama de controlo para iniciar;
3. Envio das tramas de dados;
4. Envio da trama de controlo para finalizar;
5. Fecho da ligação entre os computadores.

No caso do receptor, a sequência de acontecimentos seria a seguinte:

1. Abertura da ligação entre os computadores;
2. Receção da trama de controlo para iniciar;
3. Receção das tramas de dados;
4. Receção da trama de controlo para finalizar;
5. Fecho da ligação entre os computadores.

6 Protocolo de ligação lógica

O protocolo de ligação lógica é responsável pelas interações com a porta-série, tais como a abertura, o fecho, a leitura e a escrita de dados e pelo tratamento de tramas (delineamento, stuffing, proteção e retransmissão). De seguida estão explicadas as funções principais desta camada.

6.1 llopen

Esta função tem como objetivo iniciar a comunicação entre os dois computadores através da porta-série. Para tal, começa por chamar a função open_serial_port que se encontra no ficheiro physical_layer. Nesta, a porta-série é aberta com as flags de leitura e escrita e o VTIME é definido para o valor de 1 e o VMIN para 0. Dependendo do tipo de função (transmissor ou receptor), irão invocar SET_transmitter ou SET_receiver. No primeiro caso, a mensagem de SET vai ser enviada e vai esperar pela mensagem de UA. No caso do receptor, a mensagem inicial é lida (neste caso de SET) e após a leitura, envia a mensagem UA.

6.2 llwrite

A função, inicialmente, cria uma frame, com um cabeçalho, para as mensagens e de seguida invoca a função stuffing, que encapsula a informação das tramas. Depois a função permite a escrita de dados, dentro dos limites do timeout e das retransmissions.

6.3 llread

A função lê a mensagem, de seguida verifica a existência de erros no BCC1 e no BCC2, que indicam se a transmissão foi realizada de forma correta. De seguida, o destuffing ocorre, a frame de supervisão é removida e são enviadas as mensagens RR ou REJ. RR no caso de sucesso e REJ no caso de haver um erro na transmissão.

6.4 llclose

A função, no caso de ser um transmissor, recebe a mensagem DISC e escreve a mensagem UA para avisar o receptor de que a conexão foi terminada. No caso de ser um receptor envia a mensagem DISC e espera pela mensagem UA. No final é invocada a função close_serial_port do ficheiro physical_layer que fecha a porta-série.

7 Protocolo de aplicação

O protocolo de aplicação é responsável pelo envio e receção dos ficheiros, processamento do serviço (tratamento de cabeçalhos e distinção entre pacotes de controlo e de dados). Assim podemos verificar que foram enviadas tramas de controlo de início e fim, com o tamanho do ficheiro e nome.

8 Validação

Para testar o funcionamento do programa, este foi sujeito a diferentes testes, tais como:

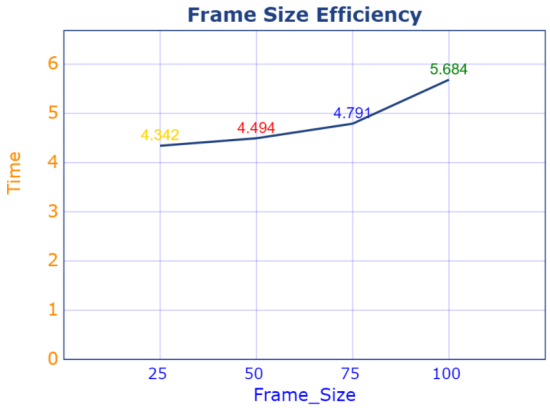
- Envio de ficheiros de tamanhos diferentes;
- Envio do mesmo ficheiro com pacotes de tamanhos diferentes;
- Envio do mesmo ficheiro com timeouts diferentes;
- Envio do mesmo ficheiro com retransmissions diferentes;
- Interrupção da ligação da porta-série durante o envio do ficheiro;
- Introdução de ruído na porta-série durante o envio do ficheiro.

O programa concluiu os testes descritos em cima com sucesso, à exceção do teste de ruído, que falhou em alguns casos devido a um problema na alocação de memória.

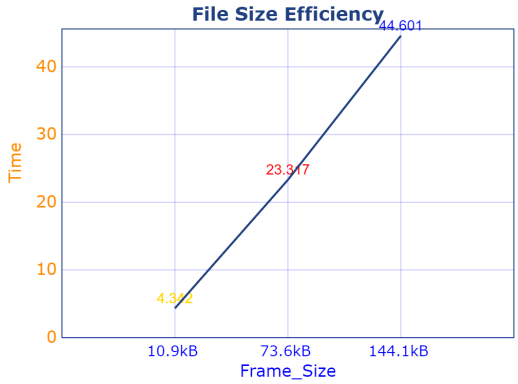
9 Eficiência do protocolo de ligação de dados

Todos os testes foram realizados nos computadores dos laboratórios, quer presencialmente quer por ligação SSH, para que os resultados obtidos fossem o mais viável possível. Os gráficos seguintes mostram os resultados obtidos nos testes de tempo usando o comando time do Linux. Os testes efetuados foram respetivamente variação da eficiência relativamente ao tamanho dos ficheiros e variação da eficiência relativamente ao tamanho da trama de dados (o teste de variação da eficiência relativamente à capacidade de ligação não foi possível uma vez que estes testes foram realizados após a aula prática, assim, não foi possível alterar o baudrate).

9.1 Variação da eficiência relativamente ao tamanho da trama de dados



9.2 Variação da eficiência relativamente ao tamanho dos ficheiros



Tendo em conta os gráficos anteriores, podemos concluir que, quanto menor o tamanho da trama de dados pior será a eficiência do protocolo de ligação dados. Contrariamente ao gráfico discutido anteriormente, o da variação da eficiência relativamente ao tamanho dos ficheiros mostra que a eficiência do programa aumenta com a diminuição do tamanho do ficheiro.

10 Conclusões

O trabalho laboratorial teve como objetivo a implementação de um protocolo de ligação de dados para a transferência de dados entre dois computadores através de uma porta-série. Após a sua implementação, este foi testado e avaliado relativamente à sua eficiência. Em suma, a criação do protocolo de ligação de dados foi bem sucedida, com todos os objetivos do guião cumpridos. Ao longo da realização deste trabalho fomos capazes de aprofundar conhecimentos teóricos relativamente à independência de camadas e o protocolo Stop&Wait.

A Anexo I - Código Fonte

```
OBJS = physical_layer.o datalink_layer.o application_layer.o program.o
PROGRAM = ./execute

executable: $(PROGRAM)
$(PROGRAM) : $(OBJS)
  $(CC) -o $(PROGRAM) -Wall $(OBJS)

transmitter: $(PROGRAM)
  $(PROGRAM) /dev/ttyS0 w pinguim.gif 100 3 4

receiver: $(PROGRAM)
  $(PROGRAM) /dev/ttyS0 r 3 4

clear:
  rm -f $(PROGRAM) $(OBJS)
```

1: Makefile

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <signal.h>
#include <termios.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "physical_layer.h"
#include "datalink_layer.h"
#include "application_layer.h"

int is_start = FALSE;

int main(int argc, char** argv) {
  int start_end_max_size;
  unsigned char *start_package, *end_package, *message, null_val[] = {0xAA};

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

    if(strcmp("w", argv[2]) != 0 && strcmp("r", argv[2]) != 0) {
        printf("Usage:\tinvalid read/write mode. a correct mode (w/r).\n\tex
    : nserial /dev/ttyS0 w\n");
        exit(1);
    }

  srand(time(NULL));
  application.status = argv[2];

```

```
34  if(strcmp("w", application.status) == 0) {
35    if(argv[3] == NULL || argv[4] == NULL) {
36      printf("You need to specify the file and the size to read\n");
37      exit(1);
38    }
39
40    if(argv[5] == NULL || argv[6] == NULL) {
41      printf("You need to specify the timeout the maximum of retransmissions
      \n");
42      exit(1);
43    }
44
45    application.file_descriptor = llopen(argv[1], application.status, atoi(
      argv[5]), atoi(argv[6]));
46
47    if(application.file_descriptor > 0) {
48      file.file_name = (char*) argv[3];
49      file.read_size = atoi(argv[4]);
50
51      file.fp = fopen((char*) file.file_name, "rb");
52
53      if(file.fp == NULL) {
54          printf("Invalid file!\n");
55          exit(-1);
56      }
57
58            if((file.file_size = file_size()) == -1)
59        return FALSE;
60
61            start_end_max_size = 2 * (strlen(file.file_name) + 9 ) +
      MAX_SIZE_LINK; //max size for start/end package;
62      start_package = malloc(start_end_max_size * sizeof(unsigned char));
63
64      int start_created_size = START_END_package(start_package,
      START_PACKAGE_TYPE);
65
66      if(start_created_size == -1) {
67                printf("Error creating start packet\n");
68        exit(-1);
69      }
70
71      is_start = TRUE;
72
73            if(send_message(start_package, start_created_size) == FALSE)
74        llclose(application.file_descriptor, -1);
75
76      readfile();
77
78      is_start = TRUE;
79      end_package = malloc(start_end_max_size * sizeof(unsigned char));
80
81      int end_package_size = START_END_package(end_package, END_PACKAGE_TYPE
      );
82
83      is_start = TRUE;
84
85            if(send_message(end_package, end_package_size) == FALSE)
86        llclose(application.file_descriptor, -1);
87
88      llclose(application.file_descriptor, TRANSMITTER);
89      }
```

```
90      }
91
92      else if(strcmp("r", application.status) == 0) {
93        if(argv[3] == NULL || argv[4] == NULL) {
94            printf("You need to specify the timeout the maximum of
    retransmissions\n");
95            exit(1);
96        }
97
98        application.file_descriptor = llopen(argv[1], application.status, atoi
    (argv[3]), atoi(argv[4]));
99
100             if(application.file_descriptor > 0) {
101         do {
102           message = get_message();
103
104           if(message == NULL)
105             message = null_val;
106
107         } while(message[0] != DISC);
108       }
109     }
110
111       else {
112       printf("Error opening serial port!\n");
113       return 1;
114     }
115
116   return 0;
117 }
```

2: program.c

```
1  #ifndef APPLICATION_LAYER_H
2  #define APPLICATION_LAYER_H
3
4  #define DATA_CONTROL 1
5
6  #define MAX_SIZE_LINK 7 // 2 * 1(BCC2) + 5 other element trama datalink
7
8  #define START_PACKAGE_TYPE 1
9  #define END_PACKAGE_TYPE 0
10
11 typedef struct {
12   int file_size;
13   char* file_name;
14   FILE* fp;
15   int read_size;
16 } File;
17
18 typedef struct {
19   int file_descriptor;
20   char* status;
21 } Application;
22
```

```
23 File file;
24 Application application;
25
26 int send_message(unsigned char* message, int length);
27
28 unsigned char* get_message();
29
30 unsigned char* get_only_data(unsigned char* message_read, int* length);
31
32 unsigned char* data_package(unsigned char* message, int* length);
33
34 int START_END_package(unsigned char* package, int type);
35
36 int file_size();
37
38 void file_parameters(unsigned char* message);
39
40 void readfile();
41
42 void writefile(unsigned char* data, int read_size);
43
44 int verify(unsigned char* message);
45
46 #endif
```

3: application_layer.h

```
1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <fcntl.h>
4 #include <unistd.h>
5 #include <signal.h>
6 #include <termios.h>
7 #include <stdio.h>
8 #include <stdlib.h>
9 #include <string.h>
10 #include "physical_layer.h"
11 #include "datalink_layer.h"
12 #include "application_layer.h"
13
14 extern int is_start;
15
16 int send_message(unsigned char* message, int length) {
17   int res;
18
19   if(is_start == TRUE) {
20     is_start = FALSE;
21     res = llwrite(application.file_descriptor, message, &length);
22   }
23
24   else {
25     unsigned char* data_package_ = data_package(message, &length);
26     res = llwrite(application.file_descriptor, data_package_, &length);
27   }
28
```

```
29    if(res == FALSE)
30      return FALSE;
31
32    return TRUE;
33  }
34
35  unsigned char* get_message() {
36    int length;
37    unsigned char *message_read = llread(application.file_descriptor, &length)
        , *only_data;
38    static int file_received_size = 0;
39
40    if(message_read == NULL || message_read[0] == DISC)
41      return message_read;
42
43    switch(message_read[0]) {
44      case 0x02:
45        file_parameters(message_read);
46        break;
47
48      case 0x01:
49        only_data = get_only_data(message_read, &length);
50        writefile(only_data, length);
51        file_received_size += length;
52        break;
53
54      case 0x03:
55        verify(message_read);
56        break;
57    }
58
59    return message_read;
60  }
61
62  unsigned char* get_only_data(unsigned char* message_read, int* length) {
63    unsigned int size = message_read[2] * 256 + message_read[3];
64    unsigned char* only_data = malloc(size * sizeof(unsigned char));
65
66      for(int j = 0; j < size; j++)
67        only_data[j] = message_read[j + 4];
68
69      *length = size;
70
71      free(message_read);
72
73      return only_data;
74  }
75
76  unsigned char* data_package(unsigned char* message, int* length) {
77    unsigned char* data_package = malloc((*length + 4) * sizeof(unsigned char)
      ), c = 0x01;
78      static unsigned int n = 0;
79      int l2 = *length / 256, l1 = *length % 256;
80
81      data_package[0] = c;
82      data_package[1] = (char) n;
83      data_package[2] = l2;
84      data_package[3] = l1;
85
86      n++;
87      n = (n % 256);
```

```
88
89      for(int i = 0; i < *length; i++)
90        data_package[i + 4] = message[i];
91
92      *length = *length + 4;
93
94      return data_package;
95  }
96
97  int START_END_package(unsigned char* package, int type) {
98    int i = 0, j = 3;
99    unsigned char file_size_char[4];
100   unsigned int file_name_length = (unsigned int) strlen(file.file_name);
101
102   //convert file size to a unsigned char array
103   file_size_char[0] = (file.file_size >> 24) & 0xFF;
104   file_size_char[1] = (file.file_size >> 16) & 0xFF;
105   file_size_char[2] = (file.file_size >> 8) & 0xFF;
106   file_size_char[3] = file.file_size & 0xFF;
107
108   //size of file size unsigned char array, normally is 4
109   int length_file_size = sizeof(file_size_char) / sizeof(file_size_char[0]);
110
111   if(type == START_PACKAGE_TYPE)
112     package[0] = 0x02;
113
114   else if(type == END_PACKAGE_TYPE)
115     package[0] = 0x03;
116
117   else
118     return -1;
119
120   package[1] = 0x00;
121   package[2] = length_file_size;
122
123   //put file size unsigned char array in package array
124   for(; i < length_file_size; i++,j++)
125     package[j] = file_size_char[i];
126
127   package[j] = 0x01;
128   j++;
129   package[j] =  file_name_length;
130   j++;
131
132   for(i = 0; i < file_name_length; i++,j++)
133     package[j] = file.file_name[i];
134
135   return j;
136 }
137
138 int file_size() {
139   fseek(file.fp, 0L, SEEK_END);
140
141     int file_size = (int) ftell(file.fp);
142
143     if(file_size == -1)
144     return -1;
145
146     fseek(file.fp, 0L, SEEK_SET);
147
148     return file_size;
```

```
149 }
150
151 void file_parameters(unsigned char* message) {
152   int i = 0, file_name_size;
153   unsigned char file_size[4];
154
155   if(message[1] == 0x00) {
156     int file_size_ = message[2];
157
158     for(; i < file_size_; i++)
159       file_size[i] = message[i + 3];
160
161     file.file_size = (file_size[0] << 24) | (file_size[1] << 16) | (
      file_size[2] << 8) | (file_size[3]);
162   }
163
164   i += 3;
165
166     if(message[i] == 0x01) {
167     i++;
168     file_name_size = message[i];
169     i++;
170     file.file_name = malloc ((file_name_size + 1) * sizeof(char));
171
172         for(int j = 0; j < file_name_size; j++, i++)
173       file.file_name[j] = message[i];
174
175     file.file_name[file_name_size] = '\0';
176   }
177
178   file.fp = fopen((char*) file.file_name, "wb");
179 }
180
181 void readfile() {
182   unsigned char* data = malloc(file.read_size * sizeof(unsigned char));
183   int file_sent_size = 0;
184
185   fseek(file.fp, 0, SEEK_SET);
186
187   while(TRUE) {
188     int res = 0;
189     res = fread(data,sizeof(unsigned char), file.read_size, file.fp);
190
191         if(res > 0) {
192       if(send_message(data,res) == FALSE) {
193         llclose(application.file_descriptor, -1);
194         exit(-1);
195       }
196
197       file_sent_size += res;
198     }
199
200     if(feof(file.fp))
201       break;
202   }
203 }
204
205 void writefile(unsigned char* data, int read_size) {
206   fseek(file.fp, 0, SEEK_END);
207   fwrite(data, sizeof(unsigned char), read_size, file.fp);
208 }
```

```
209
210 int verify(unsigned char* message) {
211   int file_size_size = message[2], file_size_total;
212   unsigned char file_size_[4];
213
214   for(int i = 0; i < file_size_size; i++)
215     file_size_[i] = message[i + 3];
216
217   file_size_total = (file_size_[0] << 24) | (file_size_[1] << 16) | (
      file_size_[2] << 8) | (file_size_[3]);
218
219   if(file_size_total == file.file_size && file_size_total == file_size())
220         return TRUE;
221
222   else
223     return FALSE;
224
225   return FALSE;
226 }
```

4: application_layer.c

```
1 #ifndef DATALINK_LAYER_H
2 #define DATALINK_LAYER_H
3
4 #define FALSE 0
5 #define TRUE 1
6
7 #define RECEIVER 0
8 #define TRANSMITTER 1
9
10 #define FLAG 0x7E
11 #define ADDRESS_T 0x03
12 #define ADDRESS_R 0x01
13 #define CONTROL_T 0x03
14 #define CONTROL_R 0x07
15 #define BCC_T 0x00
16 #define BCC_R 0x04
17
18 #define CONTROL_START 2
19 #define CONTROL_END 3
20
21 #define FRAME_S 0
22 #define FRAME_I 1
23
24 #define REJ 0
25 #define RR 1
26
27 #define DISC 0x0B
28
29 #define ERROR_PERCENTAGE_BCC1 0
30 #define ERROR_PERCENTAGE_BCC2 0
31
32 #define S0 0
33 #define S1 1
```

```
34 #define S2 2
35 #define S3 3
36 #define S4 4
37 #define SESC 5
38
39 typedef struct {
40   struct termios oldtio;
41   struct termios newtio;
42   unsigned char control_value;
43   unsigned int timeout;
44   unsigned int max_retransmissions;
45 } Datalink;
46
47 Datalink datalink;
48
49 void alarm_handler();
50
51 void state_machine(unsigned char c, int* state, unsigned char* frame, int*
       length, int frame_type);
52
53 int SET_transmitter(int* fd);
54
55 int SET_receiver(int* fd);
56
57 int llopen(char* port, char* mode, int timeout, int max_retransmissions);
58
59 int llwrite(int fd, unsigned char* message, int* length);
60
61 unsigned char* llread(int fd, int* length);
62
63 void llclose(int fd, int type);
64
65 unsigned char* create_frame(unsigned char* message, int* length);
66
67 unsigned char* frame_header(unsigned char* stuffed_frame, int* length);
68
69 unsigned char* remove_supervision_frame(unsigned char* message, int* length)
       ;
70
71 unsigned char* stuffing(unsigned char* message, int* length);
72
73 unsigned char* destuffing(unsigned char* message, int* length);
74
75 unsigned char* BCC2(unsigned char* control_message, int* length);
76
77 unsigned char* BCC1_random_error(unsigned char* package, int size_package);
78
79 unsigned char* BCC2_random_error(unsigned char* package, int size_package);
80
81 int send_RR_REJ(int fd, unsigned int type, unsigned char c);
82
83 unsigned char* send_DISC(int fd);
84
85 #endif
```

5: datalink_layer.h

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <signal.h>
#include <termios.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "physical_layer.h"
#include "datalink_layer.h"

int STOP = FALSE;
unsigned char flag_attempts = 1, flag_alarm = 1, flag_error = 0, duplicate = FALSE, control_values[] = {0x00, 0x40, 0x05, 0x85, 0x01, 0x81};

void alarm_handler() {
  flag_attempts++;

  printf("Alarm #%d\n", flag_attempts);

    if(flag_attempts >= datalink.max_retransmissions)
    flag_error = 1;

  flag_alarm = 1;
}

void state_machine(unsigned char c, int* state, unsigned char* frame, int *length, int frame_type) {
  switch(*state) {
    case S0:
      if(c == FLAG) {
        *state = S1;
        frame[*length - 1] = c;
      }
      break;

        case S1:
      if(c != FLAG) {
        frame[*length - 1] = c;

                if(*length == 4) {
          if((frame[1] ^ frame[2]) != frame[3])
            *state = SESC;

          else
            *state = S2;
        }
      }

      else {
        *length = 1;
        frame[*length - 1] = c;
      }
      break;

        case S2:
      frame[*length - 1] = c;

            if(c == FLAG) {
        STOP = TRUE;
```

```
60          alarm(0);
61          flag_alarm = 0;
62        }
63
64              else {
65          if(frame_type == FRAME_S){
66            *state = S0;
67            *length = 0;
68          }
69        }
70        break;
71
72      case SESC:
73        frame[*length - 1] = c;
74
75              if(c == FLAG) {
76          if(frame_type == FRAME_I){
77            flag_error = 1;
78            STOP = TRUE;
79          }
80
81                  else{
82            *state = S0;
83            *length = 0;
84          }
85        }
86    }
87  }
88
89  int SET_transmitter(int* fd) {
90      unsigned char SET[5] = {FLAG, ADDRESS_T, CONTROL_T, BCC_T, FLAG}, elem,
      frame[5];
91      int res, frame_length = 0, state = S0;
92
93      (void) signal(SIGALRM, alarm_handler);
94
95      while(flag_alarm == 1 && datalink.max_retransmissions > flag_attempts) {
96          res = write(*fd, SET, 5);
97
98          alarm(datalink.timeout);
99          flag_alarm = 0;
100
101         //Wait for UA signal.
102
103         while(flag_alarm == 0 && STOP == FALSE) {
104             res = read(*fd, &elem, 1);
105
106             if(res > 0) {
107                 frame_length++;
108               state_machine(elem, &state, frame, &frame_length, FRAME_S);
109           }
110       }
111   }
112
113   if(flag_error == 1)
114      return FALSE;
115
116   else
117     return TRUE;
118 }
119
```

```
120 int SET_receiver(int* fd) {
121     unsigned char UA[5] = {FLAG, ADDRESS_T, CONTROL_R, BCC_R, FLAG}, elem,
    frame[5];
122     int res, frame_length = 0, state = S0;
123 
124     while(STOP == FALSE) {
125         res = read(*fd, &elem, 1);
126 
127         if(res > 0) {
128         frame_length++;
129             state_machine(elem, &state, frame, &frame_length, FRAME_S);
130       }
131     }
132 
133   res = write(*fd, UA, 5);
134 
135   return TRUE;
136 }
137 
138 int llopen(char* port, char* mode, int timeout, int max_retransmissions) {
139   int fd, result;
140 
141   datalink.control_value = 0;
142   datalink.timeout = timeout;
143   datalink.max_retransmissions= max_retransmissions;
144 
145   open_serial_port(port, &fd);
146 
147   if(strcmp(mode, "w") == 0)
148     result = SET_transmitter(&fd);
149 
150   else if(strcmp(mode, "r") == 0)
151     result = SET_receiver(&fd);
152 
153   if(result == TRUE)
154   return fd;
155 
156   else {
157   llclose(fd, -1);
158   return -1;
159   }
160 }
161 
162 int llwrite(int fd, unsigned char* message, int* length) {
163   unsigned char* full_message = create_frame(message, length), elem, frame
      [5];
164   int res, frame_length = 0, state = S0;
165 
166   if(*length < 0)
167     return FALSE;
168 
169   flag_attempts = 1;
170   flag_alarm = 1;
171   flag_error = 0;
172   STOP = FALSE;
173 
174   while(flag_alarm == 1 && datalink.max_retransmissions > flag_attempts) {
175     res = write(fd, full_message, *length);
176     alarm(datalink.timeout);
177     flag_alarm = 0;
178 
```

```
179         //Wait for response signal.
180
181         while(flag_alarm == 0 && STOP == FALSE) {
182       res = read(fd, &elem, 1);
183
184             if(res > 0) {
185         frame_length++;
186         state_machine(elem, &state, frame, &frame_length, FRAME_S);
187       }
188     }
189
190     if(STOP == TRUE) {
191       if(control_values[datalink.control_value + 4] == frame[2]) {
192         flag_alarm = 1;
193         flag_attempts = 1;
194         flag_error = 0;
195         STOP = FALSE;
196         state = S0;
197         frame_length = 0;
198       }
199     }
200   }
201
202   if (flag_error == 1)
203     return FALSE;
204
205   datalink.control_value = datalink.control_value ^ 1;
206
207     return TRUE;
208 }
209
210 unsigned char* llread(int fd, int* length) {
211     unsigned int message_array_length = 138;
212   unsigned char elem, *message = malloc(message_array_length * sizeof(
      unsigned char)), *finish = malloc(1 * sizeof(unsigned char));
213   int res, state = S0;
214
215   *length = 0;
216   flag_error = 0;
217   STOP = FALSE;
218   finish[0] = DISC;
219
220     while(STOP == FALSE) {
221     res = read(fd, &elem, 1);
222
223         if(res > 0) {
224       *length += 1;
225
226             if(message_array_length <= *length) {
227         message_array_length *= 2;
228         message = realloc(message, message_array_length * sizeof(unsigned
      char));
229       }
230
231       state_machine(elem, &state, message, length, FRAME_I);
232     }
233   }
234
235   if(message[4] == ADDRESS_R && flag_error != 1) {
236     message = BCC1_random_error(message, *length);
237     message = BCC2_random_error(message, *length);
```

```
238   }
239
240   if(flag_error == 1 || message[2] == CONTROL_T || message[2] == CONTROL_R)
241     return NULL;
242
243   if(message[2] == DISC) {
244     llclose(fd, RECEIVER);
245     return finish;
246   }
247
248   duplicate = (control_values[datalink.control_value] == message[2]) ? FALSE
        : TRUE;
249   unsigned char temp = message[2], *no_head_message =
      remove_supervision_frame(message, length), *no_BCC2_message = BCC2(
      no_head_message, length);
250
251   if(*length == -1) {
252     if(duplicate == TRUE) {
253       send_RR_REJ(fd, RR, temp);
254       return NULL;
255     }
256
257     else {
258       send_RR_REJ(fd, REJ, temp);
259       return NULL;
260     }
261   }
262
263   else {
264     if(duplicate != TRUE) {
265       datalink.control_value = send_RR_REJ(fd, RR, temp);
266       return no_BCC2_message;
267     }
268
269     else {
270       send_RR_REJ(fd, RR, temp);
271       return NULL;
272     }
273   }
274 }
275
276 void llclose(int fd, int type) {
277   unsigned char* received, UA[5] = {FLAG, ADDRESS_T, CONTROL_R, BCC_R, FLAG
        };
278
279   if(type == TRANSMITTER) {
280     received = send_DISC(fd);
281         write(fd, UA, 5);
282         sleep(1);
283   }
284
285   else if(type == RECEIVER)
286     received = send_DISC(fd);
287
288   close_serial_port(fd);
289 }
290
291 unsigned char* create_frame(unsigned char* message, int* length) {
292   unsigned char BCC2 = 0x00, *new_message = malloc((*length + 1) * sizeof(
        unsigned char));
293   int i = 0;
```

```
294
295     for(; i < *length; i++) {
296     new_message[i] = message[i];
297     BCC2 ^= message[i];
298   }
299
300     new_message[*length] = BCC2;
301   *length += 1;
302   i = 0;
303
304   unsigned char* stuffed_message = stuffing(new_message, length), *
      control_message = frame_header(stuffed_message, length);
305
306   return control_message;
307 }
308
309 unsigned char* frame_header(unsigned char* stuffed_frame, int* length) {
310   unsigned char* full_frame = malloc((*length + 5) * sizeof(unsigned char));
311
312   full_frame[0] = FLAG;
313   full_frame[1] = ADDRESS_T;
314   full_frame[2] = control_values[datalink.control_value];
315   full_frame[3] = full_frame[1] ^ full_frame[2];
316
317     for(int i = 0; i < *length; i++)
318     full_frame[i + 4] = stuffed_frame[i];
319
320     full_frame[*length + 4] = FLAG;
321     *length +=  5;
322
323   free(stuffed_frame);
324
325   return full_frame;
326 }
327
328 unsigned char* remove_supervision_frame(unsigned char* message, int* length)
      {
329   unsigned char* control_message = malloc((*length - 5) * sizeof(unsigned
      char));
330   int i = 4, j = 0;
331
332   for(; i < *length - 1; i++, j++)
333     control_message[j] = message[i];
334
335   *length -= 5;
336
337     free(message);
338
339   return control_message;
340 }
341
342 unsigned char* stuffing(unsigned char* message, int* length) {
343   unsigned int array_length = *length;
344     unsigned char* str = malloc(array_length * sizeof(unsigned char));
345   int j = 0;
346
347     for(int i = 0; i < *length; i++, j++) {
348     if(j >= array_length) {
349       array_length *= 2;
350       str = realloc(str, array_length * sizeof(unsigned char));
351     }
```

```
352
353     if(message[i] == 0x7d) {
354       str[j] = 0x7d;
355       str[j + 1]= 0x5d;
356       j++;
357     }
358
359     else if(message[i] ==  0x7e) {
360       str[j] = 0x7d;
361       str[j + 1] = 0x5e;
362       j++;
363     }
364
365     else
366       str[j] = message[i];
367   }
368
369   *length = j;
370
371     free(message);
372
373   return str;
374 }
375
376 unsigned char* destuffing(unsigned char* message, int* length) {
377   unsigned int array_length = 133;
378   unsigned char* str = malloc(array_length * sizeof(unsigned char));
379   int new_length = 0;
380
381   for(int i = 0; i < *length; i++) {
382     new_length++;
383
384     if(new_length >= array_length) {
385       array_length *= 2;
386       str = realloc(str, array_length * sizeof(unsigned char));
387     }
388
389     if(message[i] == 0x7d) {
390       if(message[i + 1] == 0x5d) {
391         str[new_length -1] = 0x7d;
392         i++;
393       }
394
395       else if(message[i + 1] == 0x5e) {
396         str[new_length-1] = 0x7e;
397         i++;
398       }
399     }
400
401     else
402       str[new_length - 1] = message[i];
403   }
404
405   *length = new_length;
406
407     free(message);
408
409     return str;
410 }
411
412 unsigned char* BCC2(unsigned char* control_message, int* length) {
```

```
413   unsigned char control_BCC2 = 0x00, *destuffed_message = destuffing(
        control_message, length);
414   int i = 0;
415
416   for(; i < *length - 1; i++)
417     control_BCC2 ^= destuffed_message[i];
418
419   if(destuffed_message[*length - 1] != control_BCC2) {
420     *length = -1;
421     return NULL;
422   }
423
424   *length -= 1;
425   unsigned char* data_message = malloc(*length * sizeof(unsigned char));
426
427     for(i = 0; i < *length; i++)
428     data_message[i] = destuffed_message[i];
429
430   free(destuffed_message);
431
432     return data_message;
433 }
434
435 unsigned char* BCC1_random_error(unsigned char* package, int size_package) {
436   unsigned char* messed_up_message = malloc(size_package * sizeof(unsigned
        char)), letter;
437   int random = (rand() % 100) + 1;
438
439   memcpy(messed_up_message, package, size_package);
440
441   if(ERROR_PERCENTAGE_BCC1 >= random) {
442     do {
443       letter = (unsigned char) ('A' + (rand() % 256));
444     } while(letter == messed_up_message[3]);
445
446     messed_up_message[3] = letter;
447     flag_error = 1;
448
449     printf("BCC1 messed UP\n");
450   }
451
452   free(package);
453
454   return messed_up_message;
455 }
456
457
458 unsigned char* BCC2_random_error(unsigned char* package, int size_package) {
459   unsigned char* messed_up_message = malloc(size_package * sizeof(unsigned
        char)), letter;
460   int random = (rand() % 100) + 1;
461
462   memcpy(messed_up_message, package, size_package);
463
464   if(ERROR_PERCENTAGE_BCC2 >= random) {
465     //change data to have error in BCC2
466     int i = (rand() % (size_package - 5)) + 4;
467
468     do {
469       letter = (unsigned char) ('A' + (rand() % 256));
470     } while(letter == messed_up_message[i]);
```

```
471
472     messed_up_message[i] = letter;
473
474     printf("Data messed up\n");
475   }
476
477   free(package);
478
479   return messed_up_message;
480 }
481
482 int send_RR_REJ(int fd, unsigned int type, unsigned char c) {
483   unsigned char bool_val, response[5];
484
485   response[0] = FLAG;
486   response[1] = ADDRESS_T;
487   response[4] = FLAG;
488
489     if(c == 0x00)
490     bool_val = 0;
491
492     else
493     bool_val = 1;
494
495   switch (type) {
496     case RR:
497       response[2] = control_values[(bool_val ^ 1) + 2];
498       break;
499
500     case REJ:
501       response[2] = control_values[bool_val + 4];
502       break;
503   }
504
505   response[3] = response[1] ^ response[2];
506
507   write(fd, response, 5);
508
509   return bool_val ^ 1;
510 }
511
512 unsigned char* send_DISC(int fd) {
513   unsigned char elem, *frame = malloc(5 * sizeof(unsigned char)), disc[5] =
      {FLAG, ADDRESS_T, DISC, ADDRESS_T ^ DISC, FLAG};
514   int res, frame_length = 0, state = 0;
515
516     flag_attempts =1;
517   flag_alarm = 1;
518   flag_error = 0;
519   STOP = FALSE;
520
521   while(flag_alarm == 1 && datalink.max_retransmissions > flag_attempts) {
522         res = write(fd, disc, 5);
523         alarm(datalink.timeout);
524     flag_alarm = 0;
525
526     while(flag_alarm == 0 && STOP == FALSE) {
527             res = read(fd,&elem,1);
528
529             if(res > 0) {
530                 frame_length++;
```

```
531             state_machine(elem, &state, frame, &frame_length, FRAME_S);
532           }
533         }
534       }
535
536       return frame;
537     }
```

6: datalink_layer.c

```
1  #ifndef PHYSICAL_LAYER_H
2  #define PHYSICAL_LAYER_H
3
4  #define BAUDRATE B38400
5  #define MODEMDEVICE "/dev/ttyS1"
6  #define _POSIX_SOURCE 1 /* POSIX compliant source */
7
8  void open_serial_port(char* port, int* fd);
9
10 int close_serial_port(int fd);
11
12 #endif
```

7: physical_layer.h

```
1  #include <sys/types.h>
2  #include <sys/stat.h>
3  #include <fcntl.h>
4  #include <unistd.h>
5  #include <signal.h>
6  #include <termios.h>
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include <string.h>
10 #include "physical_layer.h"
11 #include "datalink_layer.h"
12
13 /* SET Serial Port Initilizations */
14 void open_serial_port(char* port, int* fd) {
15     /*
16       Open serial port device for reading and writing and not as controlling
       tty
17       because we don't want to get killed if linenoise sends CTRL-C.
18     */
19
20     *fd = open(port, O_RDWR | O_NOCTTY );
21
22   if(*fd < 0) {
23         perror(port);
```

```
24          exit(-1);
25      }
26
27      if(tcgetattr(*fd, &datalink.oldtio) == -1) { /* save current port
        settings */
28        perror("tcgetattr");
29        exit(-1);
30      }
31
32      bzero(&datalink.newtio, sizeof(datalink.newtio));
33      datalink.newtio.c_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;
34      datalink.newtio.c_iflag = IGNPAR;
35      datalink.newtio.c_oflag = 0;
36
37      /* set input mode (non-canonical, no echo,...) */
38      datalink.newtio.c_lflag = 0;
39
40      datalink.newtio.c_cc[VTIME] = 1;   /* inter-character timer unused */
41      datalink.newtio.c_cc[VMIN] = 0;   /* blocking read until 1 char received
         */
42
43      tcflush(*fd, TCIOFLUSH);
44
45      if(tcsetattr(*fd,TCSANOW, &datalink.newtio) == -1) {
46        perror("tcsetattr");
47        exit(-1);
48      }
49  }
50
51  int close_serial_port(int fd) {
52    if(tcsetattr(fd,TCSANOW, &datalink.oldtio) == -1) {
53        perror("tcsetattr");
54        exit(-1);
55      }
56
57      close(fd);
58
59      return 0;
60  }#include <sys/types.h>
61  #include <sys/stat.h>
62  #include <fcntl.h>
63  #include <unistd.h>
64  #include <signal.h>
65  #include <termios.h>
66  #include <stdio.h>
67  #include <stdlib.h>
68  #include <string.h>
69  #include "physical_layer.h"
70  #include "datalink_layer.h"
71
72  /* SET Serial Port Initilizations */
73  void open_serial_port(char* port, int* fd) {
74      /*
75        Open serial port device for reading and writing and not as controlling
         tty
76        because we don't want to get killed if linenoise sends CTRL-C.
77      */
78
79      *fd = open(port, O_RDWR | O_NOCTTY );
80
81    if(*fd < 0) {
```

```
82          perror(port);
83          exit(-1);
84      }
85
86      if(tcgetattr(*fd, &datalink.oldtio) == -1) { /* save current port
        settings */
87        perror("tcgetattr");
88        exit(-1);
89      }
90
91      bzero(&datalink.newtio, sizeof(datalink.newtio));
92      datalink.newtio.c_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;
93      datalink.newtio.c_iflag = IGNPAR;
94      datalink.newtio.c_oflag = 0;
95
96      /* set input mode (non-canonical, no echo,...) */
97      datalink.newtio.c_lflag = 0;
98
99      datalink.newtio.c_cc[VTIME] = 1;   /* inter-character timer unused */
100     datalink.newtio.c_cc[VMIN] = 0;   /* blocking read until 1 char received
         */
101
102     tcflush(*fd, TCIOFLUSH);
103
104     if(tcsetattr(*fd,TCSANOW, &datalink.newtio) == -1) {
105       perror("tcsetattr");
106       exit(-1);
107     }
108 }
109
110 int close_serial_port(int fd) {
111   if(tcsetattr(fd,TCSANOW, &datalink.oldtio) == -1) {
112       perror("tcsetattr");
113       exit(-1);
114     }
115
116     close(fd);
117
118     return 0;
119 }
```

8: physical_layer.c