Protocolo de Ligação de Dados

Relatório

Universidade do Porto

Faculdade de Engenharia

FEUP

Redes de Computadores

3^o ano

Mestrado Integrado em Engenharia Informática e Computação

Turma 4

Carolina Moreira	201303494	up201303494@fe.up.pt
Daniel Fazeres	201502846	up201502846@fe.up.pt
José Peixoto	200603103	ei12134@fe.up.pt

10 de Novembro de 2016

Conteúdo

Resumo

No âmbito da unidade curricular de Redes de Computadores, foi-nos proposto o desenvolvimento de uma aplicação que testasse um protocolo de ligação de dados criado de raiz, transferindo um ficheiro recorrendo à porta de série $RS-232$. O trabalho permitiu praticar conceitos teóricos no desenho de um protocolo de ligação de dados como o sincronismo e delimitação de tramas, controlo de erros, controlo de fluxo recurso a mecanismos de transparência de dados na transmissão assíncrona.

Findo o projeto, notou-se a importância dos mecanismos que asseguram tolerância a falhas fornecidos pela camada de ligação de dados, uma vez que a camada física não é realmente fiável.

1 Introdução

O objetivo do trabalho realizado nas aulas laboratoriais da disciplina de Redes de Computadores é a implementação de um protocolo de ligação de dados que permita praticar conhecimentos acerca de transmissões de dados entre computadores, programando em baixo nível as características comuns a este tipos de protocolos como a transparência na transmissão de dados de forma assíncrona e organização da informação sob a forma de tramas.

Este relatório pretende explicar o projeto final descrevendo a sua estrutura e os principais casos de uso.

2 Arquitetura

O projeto está organizado em duas camadas principais: a camada de aplicação e a camada de ligação de dados. As camadas respeitam o princípio de independência uma vez que cada uma apenas se responsabiliza/conhece um tipo de tarefa específica, no caso da camada de aplicação, de mais alto nível, lida-se com a interação de ficheiros e pacotes de dados e no caso da ligação de dados são feitas as tarefas de mais baixo nível relacionadas com o processamento de tramas e a interação com a porta de série.

A camada de aplicação está implementada nos ficheiros `netlink.c`, `file.c` e `packets.c`.

A camada de ligação de dados está implementada nos ficheiros `serial_port.c` `data_link_io.c` e `data_link.c`.

3 Casos de uso principais

A utilização do programa divide-se em dois propósitos distintos: envio ou receção de um ficheiro. A fase de tratamento dos parâmetros opcionais passados pela linha de comandos é comum em ambos os casos e engloba a chamada de funções como: `parse_args`, `parse_serial_port_arg`, `parse_flags` e chamadas opcionais a outras funções consoante os parâmetros passados.

Envio de ficheiro

Após a interpretação dos parâmetros opcionais e a leitura do ficheiro, o programa invoca a função `send_file` da camada de aplicação para o envio de um ficheiro que por sua vez pede à camada de ligação de dados que estabeleça uma ligação pela porta de série na chamada à função `transmitter_connect` e transmita dados através da função `transmitter_write` e termine a ligação com a chamada `disconnect`.

A chamada `transmitter_connect` da camada da ligação de dados abre a porta de série envia a trama SET e recebe a trama UA. No envio de tramas que requerem confimação de receção é usada a função `f_send_acknowledged_frame`. Antes da chamada à função `transmitter_write`, o programa organiza a informação a enviar sob a forma de pacotes de dados nas funções `send_control_packet` ou `send_data_packets`.

Receção de ficheiro

Após a interpretação dos parâmetros opcionais e a leitura do ficheiro, o programa invoca a função `receive_file` da camada de aplicação para a receção de um ficheiro que por sua vez pede à camada de ligação de dados que estabeleça uma ligação pela porta de série com a chamada à função `receiver_listen` e receba pacotes de dados através da função `receiver_read` e termine a ligação com a chamada `disconnect`.

A chamada `receiver_listen` da camada da ligação de dados abre a porta de série e espera pela receção de uma trama SET enviando em seguida a confirmação de receção com a função `f_send_frame`. Na camada de ligação de dados é usada a função `f_receive_frame` na receção de tramas. Após a receção de um pacote de dados através da função `llread`, o programa descodifica os dados recebidos nas funções `parse_control_packet` ou `parse_data_packet` caso se espere receber um pacote de controlo ou de dados respectivamente.

4 Protocolo de ligação lógica

A camada de ligação lógica é responsável pelo envio de informação através da porta de série. Para tal, uma série de mecanismos são necessários para

assegurar a transmissão com sucesso dos dados. Os dados são encapsulados em tramas com numeração e a delimitação das tramas é feita por uma sequêncial especial de oito bits. É também assegurada a transparência dos dados pela técnica de byte stuffing.

A proteção da integridade dos dados é feita pelo uso de um código detetor de erros, no caso das tramas S e U pelo $BCC1$ e nas tramas I existe um segundo código, o $BCC2$, que verifica a integridade do campo de dados.

Delimitação e envio de uma trama

```
Return_e f_send_frame(const int fd, const struct frame frame
    )
{
  byte *bp = g_buffer;

  *bp++ = FLAG;

  // write header
  *bp++ = frame.address;
  *bp++ = frame.control;
  *bp++ = frame.address ^ frame.control; // bcc

  // copy data
  if (frame.size > LL_MAX_PAYLOAD_UNSTUFFED) {
    return ERROR_CODE;

  } else if (frame.size > 0) { // frame might be 0 if it is
      supervision
    bp = copy_and_stuff_bytes(bp, frame.data, frame.size);
  }

  *bp++ = FLAG;

  if (serial_port_write(fd, g_buffer, bp - g_buffer) < 0) {
    return ERROR_CODE;
  }
  ++g_sent_frame_counter;

  return SUCCESS_CODE;
}
```

Verificação da integridade uma trama recebida

```
static Return_e parse_frame_from_array(struct frame* frame,
    byte *a)
{
  if (*a++ != FLAG) {
    return ERROR_CODE;
  }
  for (int i = 0; i <= 2; i++) { // the next fields should
      not have a FLAG
    if (a[i] == FLAG) {
```

```
    return BADFRAME_CODE;
  }
}

frame->address = *a++;
frame->control = *a++;
const byte header_bcc = *a++;
if (header_bcc != (frame->address ^ frame->control)) {
  ++g_header_bcc_error_counter;
  return BADFRAME_CODE;
}
frame->size = 0;

/* Supervision frame */
if (*a == FLAG) {
  return SUCCESS_CODE;
}

if (*(a + 1) == FLAG) {
  return BADFRAME_CODE;
}

/* Data frame */
byte data_bcc = 0;
size_t num_bytes = 0;
while (1) {
  if (num_bytes > LL_MAX_PAYLOAD_STUFFED) {
    return BADFRAME_CODE;
  }
  if (frame->size > frame->max_data_size) {
    return BADFRAME_CODE;
  }

  byte c;
  if (a[num_bytes] == BS_ESC) {
    fprintf(stderr, "----\n");
    // remove byte stuffing
    ++num_bytes;
    c = BS_OCT ^ a[num_bytes];
  } else {
    c = a[num_bytes];
  }
      ++num_bytes;
      frame->data[frame->size++] = c;
      fprintf(stderr, "%x\n", c);

  /*
   * Stop loop condition.
   */
  if (a[num_bytes] == FLAG) {
    if (frame->data[frame->size-1] != data_bcc) {
      ++g_data_bcc_error_counter;
      return BADFRAME_CODE;
    }
```

```
            return SUCCESS_CODE;
        } else {
                data_bcc ^= c;
                fprintf(stderr,"c = %x, bcc = %x\n",c,data_bcc);
            }

    }
}
```

5 Protocolo de aplicação

A camada de aplicação é responsável pela leitura/escrita dos dados do ficheiro a enviar/receber. Do lado do emissor, procede-se à segmentação do ficheiro em pacotes de dados que vão sendo numerados e enviados para a camada de ligação de dados, por forma a serem encaixados em tramas de informação e posteriormente enviados através da porta de série. Do lado do receptor, é feita a compilação e escritura dos dados recebidos num ficheiro em disco nomeado de acordo com a informação recebida nos pacotes de controlo. Quer no emissor quer no receptor, recorre-se à codificação das etapas sob a forma de máquinas de estado.

5.1 Envio de um ficheiro

A camada de aplicação pode interpretar os argumentos opcionais passados através da interface de linha de comandos para ler do disco um ficheiro para uma estrutura de dados que armazena os dados, o nome e o tamanho do ficheiro. Opcionalmente, só os dados de um ficheiro serão lidos do `stdin` para a estrutura de dados referida e será atribuído um nome de ficheiro predefinido.

```
struct file {
    const char* name;
    size_t size;
    char* data;
};
```

Após a leitura do ficheiro, a camada de aplicação entra numa máquina de estados com quatro estados ordenados: abertura de ligação, envio de pacote de controlo inicial, envio de pacotes de dados, envio de pacote de controlo final e fecho da ligação.

Método usado na abertura de ligação

```
int llopen(char *port, int transmitter);
```

Método usado para o envio de pacotes de controlo inicial e final

```
int send_control_packet(struct connection* connection,
    struct file *file,
        byte control_field);
```

Método usado para o envio de pacote de dados

```
int send_data_packets(struct connection* connection, struct
    file* file,
        size_t* num_data_bytes_sent, size_t* sequence_number
            );
```

Método usado para o envio dos pacotes para a camada de ligação de dados

```
int transmitter_write(struct connection* conn, byte*
    data_packet, size_t size);
```

Método usado no fecho da ligação

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

5.2 Receção de um ficheiro

Após a interpretação dos parâmetros passados pela linha de comandos que indicam ao programa para receber um ficheiro, a camada de aplicação entra numa máquina de estados com quatro estados ordenados: abertura de ligação, receção de pacote de controlo inicial, receção de pacotes de dados e fecho da ligação. Após o estabelecimento de uma ligação com sucesso, o programa fica à espera da receção de um pacote de controlo com os dados relativos ao tamanho e nome do ficheiro. Posteriormente, inicia-se o processo de receção dos pacotes de dados com a informação contida no ficheiro até que se receba um pacote de controlo final, sinalizando o fim da receção do ficheiro.

Método usado para receber o pacote de controlo inicial

```
int receive_start_control_packet(const int fd,
    char **file_name, size_t *file_size)
```

Método usado para descodificar um pacote de controlo

```
int parse_control_packet(const int control_packet_length,
    byte *control_packet, char **file_name,
    size_t *file_size)
```

Método usado para receber pacotes de dados e escrever em disco

```
int receive_data_packets(const int fd, char* file_name,
    size_t file_size, int attempts_left)
```

Método usado para descodificar um pacote pacote de dados

```
int parse_data_packet(const int data_packet_length,
    byte *data_packet, char **data,
    size_t* sequence_number)
```

Método usado para descodificar um pacote pacote de dados

```
int parse_data_packet(const int data_packet_length,
    byte *data_packet, char **data,
    size_t* sequence_number)
```

6 Validação

O projeto inclui testes unicamente a camada de ligação de dados nos ficheiros `data_link_test.c` e `serial_port_test`. No caso da porta de série, são feitos pequenos testes com a validação dos outputs dado um determinado input numa chamada directa das funções read e write à camada lógica.

No caso da ligação de dados é feito um teste ao nível do sistema de tramas incluíndo os códigos de verificação de integridade e byte stuffing.

7 Elementos de valorização

7.1 Selecção de parâmetros pelo utilizador

Quando o programa é invocado pela linha de comandos de forma errónea ou sem quaisquer parâmetros adicionais, são mostrados os argumentos opcionais disponíveis que permitem configurar a execução do programa, nomeadamente o modo de operação (`receiver` ou `transmitter`), leitura do ficheiro a enviar do disco ou proveniente de dados redirecionados do `stdin`, selecção da baudrate, determinação do tamanho máximo de bytes de dados enviados em cada frame e do número de tentativas na recuperação de erros.

7.2 Implementação de REJ

Quando ocorrem erros no processamento de tramas recebidas na camada de ligação de dados, é enviada de forma preemptiva ao `timeout` uma trama com confirmação negativa (`REJ`) que permite a retransmissão da trama de informação.

```
else if (ret == BADFRAME_CODE) {
  /*
  * Send 'bad frame' acknowledgment.
  */
  byte c_out = data_reply_byte(conn->frame_number, FALSE);
  if (f_send_frame(conn->fd, FRAME(c_out)) != SUCCESS_CODE)
    break;
}
```

7.3 Verificação da integridade dos dados pela Aplicação

Após receção com sucesso do primeiro pacote de controlo pela camada de aplicação, é armazenado o tamanho expectável em bytes do ficheiro a

receber, comparando-o no fim da sessão com o valor real de bytes dados recebidos. São também guardados os números de pacotes de dados perdidos e duplicados.

```
void receiver_stats()
{
    fprintf(stdout, "Receiver statistics\n");
    fprintf(stdout, "\treceived file bytes/file bytes:%zu/%
        zu\n",
            received_file_bytes, real_file_bytes);
    fprintf(stdout, "\tlost packets:%zu\n", lost_packets);
    fprintf(stdout, "\tduplicated packets:%zu\n",
        duplicated_packets);
}
```

8 Conclusões

O projeto pode ser sumariamente descrito pelo seu principal propósito que é o desenvolvimento de um protocolo de ligação de dados e o seu teste pelo sucesso na transferência de ficheiros entre dois computadores.

Referências

[1] Andrew S. Tanenbaum, David J. Wetherall, *Computer Networks*, Prentice Hall, 5th edition, 2011.

A Código fonte

A.1 Camada de aplicação

netlink.c

```
1  void receiver_stats()
2  #include <sys/types.h>
3  #include <sys/stat.h>
4  #include <fcntl.h>
5  #include <stdio.h>
6  #include <string.h>
7  #include <stdlib.h>
8  #include <termios.h>
9  #include <unistd.h>
10
11 #include "packets.h"
12 #include "file.h"
13 #include "netlink.h"
14 #include "serial_port.h"
15
16 struct file file_to_send;
```

```
17  int max_retries = 3;
18
19  void help(char **argv)
20  {
21    fprintf(stderr, "Usage: %s [OPTIONS] <serial port>\n",
        argv[0]);
22    fprintf(stderr, "\n Program options:\n");
23    fprintf(stderr, "  -t <FILEPATH>\t\ttransmit file over the
        serial port\n");
24    fprintf(stderr, "  -i\t\t\ttransmit data read from stdin\n
        ");
25    fprintf(stderr, "  -b <BAUDRATE>\t\tbaudrate of the serial
        port\n");
26    fprintf(stderr,
27        "  -p <DATASIZE>\t\tmaximum bytes of data transfered
            each frame\n");
28    fprintf(stderr, "  -r <RETRY>\t\tnumber of retry attempts\
        n");
29  }
30
31  int parse_serial_port_arg(int index, char **argv)
32  {
33    if ((strcmp("/dev/ttyS0", argv[index]) != 0)
34        && (strcmp("/dev/ttyS1", argv[index]) != 0)
35        && (strcmp("/dev/ttyS4", argv[index]) != 0)) {
36      fprintf(stderr, "Error: bad serial port value\n");
37      return -1;
38    }
39
40    return index;
41  }
42
43  int parse_baudrate_arg(int baurdate_index, char **argv)
44  {
45    if (strcmp("B50", argv[baurdate_index]) == 0) {
46      serial_port_baudrate = B50;
47      return 0;
48    } else if (strcmp("B75", argv[baurdate_index]) == 0) {
49      serial_port_baudrate = B75;
50      return 0;
51    } else if (strcmp("B110", argv[baurdate_index]) == 0) {
52      serial_port_baudrate = B110;
53      return 0;
54    } else if (strcmp("B134", argv[baurdate_index]) == 0) {
55      serial_port_baudrate = B134;
56      return 0;
57    } else if (strcmp("B150", argv[baurdate_index]) == 0) {
58      serial_port_baudrate = B150;
59      return 0;
60    } else if (strcmp("B200", argv[baurdate_index]) == 0) {
61      serial_port_baudrate = B200;
62      return 0;
63    } else if (strcmp("B300", argv[baurdate_index]) == 0) {
64      serial_port_baudrate = B300;
```

```
65      return 0;
66    } else if (strcmp("B600", argv[baurdate_index]) == 0) {
67      serial_port_baudrate = B600;
68      return 0;
69    } else if (strcmp("B1200", argv[baurdate_index]) == 0) {
70      serial_port_baudrate = B1200;
71      return 0;
72    } else if (strcmp("B1800", argv[baurdate_index]) == 0) {
73      serial_port_baudrate = B1800;
74      return 0;
75    } else if (strcmp("B2400", argv[baurdate_index]) == 0) {
76      serial_port_baudrate = B2400;
77      return 0;
78    } else if (strcmp("B4800", argv[baurdate_index]) == 0) {
79      serial_port_baudrate = B4800;
80      return 0;
81    } else if (strcmp("B9600", argv[baurdate_index]) == 0) {
82      serial_port_baudrate = B9600;
83      return 0;
84    } else if (strcmp("B19200", argv[baurdate_index]) == 0) {
85      serial_port_baudrate = B19200;
86      return 0;
87    } else if (strcmp("B38400", argv[baurdate_index]) == 0) {
88      serial_port_baudrate = B38400;
89      return 0;
90    }
91    fprintf(stderr, "Error: bad serial port baudrate value\n")
          ;
92    fprintf(stderr,
93        "Valid baudrates: B110, B134, B150, B200, B300, B600,
              B1200, B1800, B2400, B4800, B9600, B19200, B38400\
              n");
94    return -1;
95  }
96
97  void parse_max_packet_size(int packet_size_index, char **
        argv)
98  {
99    int val = atoi(argv[packet_size_index]);
100   if (val > FRAME_SIZE || val < 0)
101     max_data_transfer = FRAME_SIZE;
102   else
103     max_data_transfer = val;
104
105 #ifdef NETLINK_DEBUG_MODE
106   fprintf(stderr,"\nparse_max_packet_size:\n");
107   fprintf(stderr,"  max_packet_size=%d\n", max_data_transfer
          );
108 #endif
109 }
110
111 void parse_max_retries(int packet_size_index, char **argv)
112 {
113   int val = atoi(argv[packet_size_index]);
```

```
114   if (val <= 0)
115     max_retries = 4;
116   else
117     max_retries = 1 + val;
118
119 #ifdef NETLINK_DEBUG_MODE
120   fprintf(stderr,"\nmax_retries:\n");
121   fprintf(stderr,"  max_retries=%d\n", max_retries);
122 #endif
123 }
124
125 int parse_flags(int* t_index, int* i_index, int* b_index,
        int* p_index,
126     int* r_index, int argc, char **argv)
127 {
128   for (size_t i = 0; i < (argc - 1); i++) {
129     if ((strcmp("-t", argv[i]) == 0)) {
130       *t_index = i;
131     } else if ((strcmp("-i", argv[i]) == 0)) {
132       *i_index = i;
133     } else if ((strcmp("-b", argv[i]) == 0)) {
134       *b_index = i;
135     } else if ((strcmp("-p", argv[i]) == 0)) {
136       *p_index = i;
137     } else if ((strcmp("-r", argv[i]) == 0)) {
138       *r_index = i;
139     } else if ((argv[i][0] == '-')) {
140       return -1;
141     }
142   }
143 #ifdef NETLINK_DEBUG_MODE
144   fprintf(stderr,"\nparse_flags(): flag indexes\n");
145   fprintf(stderr,"  -t=%d\n  -i=%d\n  -b=%d\n  -p=%d\n  -r=%
        d\n", *t_index, *i_index, *b_index,
146       *p_index, *r_index);
147 #endif
148   return 0;
149 }
150
151 int parse_args(int argc, char **argv, int *is_transmitter)
152 {
153
154 #ifdef NETLINK_DEBUG_MODE
155   fprintf(stderr,"\nparse_args(): received arguments\n");
156   fprintf(stderr,"  argc=%d\n  argv=%s\n", argc, *argv);
157 #endif
158
159   if (argc < 2) {
160     return -1;
161   }
162
163   if (argc == 2)
164     return parse_serial_port_arg(1, argv);
165
```

```
166     int t_index = -1, i_index = -1, b_index = -1, p_index =
            -1, r_index = -1;
167
168     if (parse_flags(&t_index, &i_index, &b_index, &p_index, &
            r_index, argc,
169         argv)) {
170       fprintf(stderr, "Error: bad flag parameter\n");
171       return -1;
172     }
173
174     if (t_index > 0 && t_index < argc - 1) {
175       if (read_file_from_disk(argv[t_index + 1], &file_to_send
              ) < 0) {
176         return -1;
177       }
178       *is_transmitter = 1;
179     } else {
180       if (i_index > 0 && i_index < argc - 1) {
181         if (read_file_from_stdin(&file_to_send) < 0) {
182           return -1;
183         }
184         *is_transmitter = 1;
185       }
186     }
187
188     if (b_index > 0 && b_index < argc - 1) {
189       if (parse_baudrate_arg(b_index + 1, argv) != 0) {
190         return -1;
191       }
192     }
193
194     if (p_index > 0 && p_index < argc - 1) {
195       parse_max_packet_size(p_index + 1, argv);
196     }
197
198     if (r_index > 0 && r_index < argc - 1) {
199       parse_max_retries(r_index + 1, argv);
200     }
201
202     return parse_serial_port_arg(argc - 1, argv);
203   }
204
205   int main(int argc, char **argv)
206   {
207     int port_index = -1;
208     int is_transmitter = 0;
209
210     if ((port_index = parse_args(argc, argv, &is_transmitter))
            < 0) {
211       help(argv);
212       exit(EXIT_FAILURE);
213     }
214
215     if (is_transmitter) {
```

```
216       fprintf(stderr, "transmitting %s\n", file_to_send.name);
217       return send_file(argv[port_index], &file_to_send,
              max_retries);
218     } else {
219       fprintf(stderr, "receiving file\n");
220   #ifdef NETLINK_DEBUG_MODE
221       fprintf(stderr, "\tserial_port_baudrate:%d\n",
              serial_port_baudrate);
222       fprintf(stderr, "\tis_transmitter:%d\n", is_transmitter)
              ;
223   #endif
224       return receive_file(argv[port_index], max_retries);
225     }
226   }
```

file.c

```
1   #include <fcntl.h>
2   #include <stdio.h>
3   #include <stdlib.h>
4   #include <unistd.h>
5   #include <stdint.h>
6   #include <libgen.h>
7   #include <limits.h>
8   #include "file.h"
9
10  int read_file_from_stdin(struct file *f)
11  {
12    char *buffer;
13    if ((buffer = malloc(sizeof(char) * INT_MAX)) == NULL ) {
14      perror("read_file_from_stdin() buffer malloc error");
15      return -1;
16    }
17
18    size_t size = 0;
19
20    if ((size = fread(buffer, sizeof(char), INT_MAX, stdin)) <
          0) {
21      fprintf(stderr, "ERROR: reading from the stdin.\n");
22      return -1;
23    }
24
25  #ifdef APPLICATION_LAYER_DEBUG_MODE
26    fprintf(stderr, "read_file_from_stdin()\n\tname=%s\n\tsize
          =%zu\n\tdata=%s\n", "stdin.out", size,
27        buffer);
28  #endif
29
30    f->name = "output";
31    f->size = size;
32    f->data = buffer;
33
34    return 0;
35  }
```

```
36
37 int read_file_from_disk(char *name, struct file *f)
38 {
39   size_t length;
40   FILE *file = fopen(name, "r");
41
42   if (file != NULL ) {
43     fseek(file, 0L, SEEK_END);
44     length = ftell(file);
45     char *buffer = malloc(sizeof(char) * length);
46     if (buffer != NULL ) {
47       fseek(file, 0, SEEK_SET);
48       fread(buffer, 1, length, file);
49       fclose(file);
50       f->name = basename(name);
51       f->size = length;
52       f->data = buffer;
53       return 0;
54     }
55   }
56
57   fprintf(stderr, "Error: file %s is NULL.\n", name);
58   return -1;
59 }
```

packets.c

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <time.h>
4  #include <stdlib.h>
5  #include <unistd.h>
6
7  #include "data_link.h"
8  #include "byte.h"
9  #include "packets.h"
10
11 int max_data_transfer = FRAME_SIZE;
12
13 size_t real_file_bytes = 0;
14 size_t received_file_bytes = 0;
15 size_t lost_packets = 0;
16 size_t duplicated_packets = 0;
17
18 struct connection g_connections[MAX_FD];
19
20 int send_file(char *port, struct file *file, int
       max_send_attempts)
21 {
22   fprintf(stderr, "send_file\n");
23   int fd = 0;
24   struct connection* connection;
25   int attempts_left = max_send_attempts;
26   int state = SND_OPEN_CONNECTION;
```

```
27    size_t num_data_bytes_sent = 0;
28    size_t sequence_number = 0;
29
30    while (attempts_left) {
31      switch (state) {
32      case SND_OPEN_CONNECTION:
33        fprintf(stderr, "open connection\n");
34        if ((fd = llopen(port, 1)) > 0) {
35          connection = &g_connections[fd];
36          attempts_left = max_send_attempts;
37          state = SND_START_CONTROL_PACKET;
38        } else {
39  #ifdef APPLICATION_LAYER_DEBUG_MODE
40          fprintf(stderr, "llopen() returned an error code\n")
                ;
41          fprintf(stderr, "\t%d attempts left\n",
                attempts_left - 1);
42  #endif
43          state = SND_OPEN_CONNECTION;
44          retry(&attempts_left);
45        }
46        break;
47      case SND_START_CONTROL_PACKET:
48        fprintf(stderr, "start control packet\n");
49        if (send_control_packet(connection, file,
              control_field_start)
50            < 0) {
51  #ifdef APPLICATION_LAYER_DEBUG_MODE
52          fprintf(stderr,
53              "start of transmission send_start_control_packet
                    () returned an error code\n");
54          fprintf(stderr, "\t%d attempts left\n",
                attempts_left - 1);
55  #endif
56          retry(&attempts_left);
57        } else {
58          attempts_left = max_send_attempts;
59          state = SND_DATA_PACKETS;
60        }
61        break;
62      case SND_DATA_PACKETS:
63        fprintf(stderr, "send data packets\n");
64        if (send_data_packets(connection, file, &
              num_data_bytes_sent,
65            &sequence_number) < 0) {
66  #ifdef APPLICATION_LAYER_DEBUG_MODE
67          fprintf(stderr, "send_data_packets() returned an
                error code\n");
68          fprintf(stderr, "\t%d attempts left\n",
                attempts_left - 1);
69  #endif
70          retry(&attempts_left);
71        } else {
72          attempts_left = max_send_attempts;
```

```
73          state = SND_CLOSE_CONTROL_PACKET;
74        }
75        break;
76      case SND_CLOSE_CONTROL_PACKET:
77        if (send_control_packet(connection, file,
              control_field_end) < 0) {
78  #ifdef APPLICATION_LAYER_DEBUG_MODE
79          fprintf(stderr,
80              "end of transmission send_control_packet()
                    returned an error code");
81          fprintf(stderr, "\t%d attempts left\n",
                attempts_left - 1);
82  #endif
83          retry(&attempts_left);
84        } else {
85          attempts_left = max_send_attempts;
86          state = SND_CLOSE_CONNECTION;
87        }
88        break;
89      case SND_CLOSE_CONNECTION:
90        if (llclose(fd) == 0) {
91          return 0;
92        } else {
93          state = RCV_CLOSE_CONNECTION;
94          retry(&attempts_left);
95        }
96        break;
97      default:
98        return -1;
99        break;
100     }
101   }
102   return -1;
103 }
104
105 int send_control_packet(struct connection* connection,
        struct file *file,
106     byte control_field)
107 {
108   // 5 bytes plus 2 specific data type sizes (value fields)
109   size_t control_packet_size = (5 + sizeof(size_t)
110       + ((strlen(file->name) + 1) * sizeof(char)));
111   if (control_packet_size > connection->packet_size) {
112     fprintf(stderr, "control_packet_size (%zu) > (%zu)
            allowed packet size",
113         control_packet_size, connection->packet_size);
114     return -1;
115   }
116
117 #ifdef APPLICATION_LAYER_DEBUG_MODE
118   fprintf(stderr, "send_control_packet()\n");
119   fprintf(stderr, "\tcontrol_field=%d\n", control_field);
120   fprintf(stderr, "\tcontrol_packet_size=%zu\n",
        control_packet_size);
```

```
121     fprintf(stderr, "\tpacket_size=%zu\n", (connection->
            packet_size));
122     fprintf(stderr, "\tl1=%zu\n", sizeof(size_t));
123     fprintf(stderr, "\tfile_size=%zu\n", file->size);
124     fprintf(stderr, "\tname=%s\n", file->name);
125     fprintf(stderr, "\tl2=%zu\n", strlen(file->name));
126   #endif
127
128     byte* control_packet;
129     if ((control_packet = malloc(control_packet_size * sizeof(
            byte))) == NULL ) {
130       perror("send_control_packet() control_packet malloc
              error");
131       return -1;
132     }
133     control_packet[control_field_index] = control_field;
134
135     // TLV (file size)
136     size_t v1_length = sizeof(size_t);
137     control_packet[control_packet_t1_index] =
            control_packet_tlv_type_filesize;
138     control_packet[control_packet_l1_index] = v1_length;
139
140   #ifdef APPLICATION_LAYER_DEBUG_MODE
141     fprintf(stderr, "\tv1_length=%zu\n", v1_length);
142   #endif
143
144     memcpy(control_packet + control_packet_v1_index, &(file->
            size), v1_length);
145
146     // TLV (file name)
147     size_t v2_length = strlen(file->name);
148     size_t control_packet_t2_index = 4 + v1_length;
149     size_t control_packet_l2_index = control_packet_t2_index +
             1;
150     size_t control_packet_v2_index = control_packet_l2_index +
             1;
151
152     control_packet[control_packet_t2_index] =
            control_packet_tlv_type_name;
153     control_packet[control_packet_l2_index] = v2_length;
154     memcpy(control_packet + control_packet_v2_index, (byte*)
            file->name,
155            v2_length);
156   //  control_packet[control_packet_l2_index + v2_length] =
        '\0';
157
158     if (transmitter_write(connection, control_packet,
            control_packet_size)
159         < 0) {
160       free(control_packet);
161       return -1;
162     }
163
```

```
164   free(control_packet);
165   return 0;
166 }
167
168 int send_data_packets(struct connection* connection, struct
      file* file,
169     size_t* num_data_bytes_sent, size_t* sequence_number)
170 {
171   fprintf(stderr, "send_data_packets\n");
172   byte* file_data_pointer = (byte*) file->data;
173   const byte* eof_data_pointer = ((byte*) file->data
174       + file->size * sizeof(char));
175
176   for (size_t i = 0; i < *num_data_bytes_sent; i++)
177     file_data_pointer++;
178
179   while (file_data_pointer < eof_data_pointer) {
180     fprintf(stderr, "while (file_data_pointer <
          eof_data_pointer)\n");
181     size_t max_data_size = connection->packet_size
182         - data_packet_header_size;
183     if (max_data_transfer > 0 && max_data_transfer <
          max_data_size) {
184       max_data_size = max_data_transfer;
185     }
186
187     size_t remaining_data_bytes = file->size - *
          num_data_bytes_sent;
188     size_t remainder = remaining_data_bytes % (max_data_size
          );
189     size_t data_bytes_to_send =
190         remainder == 0 ? (max_data_size) : remainder;
191
192     size_t data_packet_size = data_bytes_to_send +
          data_packet_header_size;
193
194     byte* data_packet;
195     if ((data_packet = malloc(data_packet_size * sizeof(byte
          ))) == NULL ) {
196       perror("send_control_packet() data_packet malloc error
            ");
197       return -1;
198     }
199
200     data_packet[control_field_index] = control_field_data;
201     data_packet[data_packet_sequence_number_index] = (*
          sequence_number)
202         % sequence_number_modulus;
203     (*sequence_number)++;
204     data_packet[data_packet_l2_index] = (data_bytes_to_send
          / 256);
205     data_packet[data_packet_l1_index] = (data_bytes_to_send
          % 256);
```

```
206       //fprintf(stderr,"sending packet %zu\n",
              data_packet_sequence_number_index);
207       fprintf(stderr, "sequence_number: %ld\n", *
              sequence_number);
208
209       for (size_t i = 0;
210           file_data_pointer < eof_data_pointer && i <
                data_bytes_to_send;
211           i++) {
212         data_packet[i + data_packet_header_size] =
213             (byte) file->data[*num_data_bytes_sent];
214
215         file_data_pointer++;
216         (*num_data_bytes_sent)++;
217       }
218
219       if (transmitter_write(connection, data_packet,
              data_packet_size) < 0) {
220         free(data_packet);
221         fprintf(stderr, "transmitter write returned negative\n
                ");
222         return -1;
223       }
224
225       free(data_packet);
226     }
227     return 0;
228   }
229
230   int receive_file(char *port, int max_receive_attempts)
231   {
232     int fd = 0;
233     int attempts_left = max_receive_attempts;
234     char *file_name;
235     size_t file_size;
236     int state = RCV_OPEN_CONNECTION;
237
238     while (attempts_left) {
239       switch (state) {
240       case RCV_OPEN_CONNECTION:
241         fprintf(stderr, "opening connection..\n");
242         if ((fd = llopen(port, 0)) > 0) {
243           state = RCV_START_CONTROL_PACKET;
244           attempts_left = max_receive_attempts;
245         } else {
246           retry(&attempts_left);
247         }
248         break;
249
250       case RCV_START_CONTROL_PACKET:
251         fprintf(stderr, "expecting control packet\n");
252         if (receive_start_control_packet(fd, &file_name, &
                file_size) < 0) {
253   #ifdef APPLICATION_LAYER_DEBUG_MODE
```

```
254           fprintf(stderr,
255               "receive_start_control_packet() returned an
                      error code\n");
256           fprintf(stderr, "\t%d attempts left\n",
                  attempts_left - 1);
257   #endif
258           retry(&attempts_left);
259           break;
260         } else {
261           state = RCV_DATA_PACKETS;
262           attempts_left = max_receive_attempts;
263           real_file_bytes = file_size;
264         }
265         break;
266
267       case RCV_DATA_PACKETS:
268         fprintf(stderr, "expecting data packet\n");
269         if (receive_data_packets(fd, file_name, file_size,
                attempts_left)
270             < 0) {
271           receiver_stats();
272           return -1;
273           break;
274         } else {
275           state = RCV_CLOSE_CONNECTION;
276           attempts_left = max_receive_attempts;
277         }
278         break;
279
280       case RCV_CLOSE_CONNECTION:
281         if (llclose(fd) == 0) {
282           receiver_stats();
283           return 0;
284         } else {
285           state = RCV_CLOSE_CONNECTION;
286           retry(&attempts_left);
287         }
288         break;
289       default:
290         receiver_stats();
291         return -1;
292       }
293     }
294     return -1;
295   }
296
297   int receive_start_control_packet(const int fd, char **
          file_name,
298       size_t *file_size)
299   {
300     byte *control_packet;
301     int control_packet_length = 0;
302
```

```
303   if ((control_packet_length = llread(fd, &control_packet))
          < 0) {
304     free(control_packet);
305     return -1;
306   }
307
308   byte control_field = control_packet[control_field_index];
309   if (control_field == control_field_start) {
310     return parse_control_packet(control_packet_length,
            control_packet,
311          file_name, file_size);
312   }
313
314 #ifdef APPLICATION_LAYER_DEBUG_MODE
315   fprintf(stderr, "receive_data_packet(): bad control field
          value\n");
316 #endif
317
318   free(control_packet);
319   return -1;
320 }
321
322 int receive_data_packets(const int fd, char* file_name,
        size_t file_size,
323     int attempts_left)
324 {
325   FILE* received_file = fopen(file_name, "w");
326   int receive_return_value = 1;
327   size_t sequence_number = 0;
328
329 #ifdef APPLICATION_LAYER_DEBUG_MODE
330   fprintf(stderr, "receive_data_packets()\n");
331   fprintf(stderr, "\tfile_name=%s\n", file_name);
332   fprintf(stderr, "\tfile_size=%zu\n", file_size);
333 #endif
334
335   while (receive_return_value > 0 && attempts_left > 0) {
336
337     char *file_data;
338
339     if ((receive_return_value = receive_data_packet(fd, &
            file_data,
340          received_file_bytes, &sequence_number)) < 0) {
341 #ifdef APPLICATION_LAYER_DEBUG_MODE
342       fprintf(stderr, "receive_data_packet() returned an
              error code\n");
343       fprintf(stderr, "\t%d attempts left\n", attempts_left
              - 1);
344 #endif
345       retry(&attempts_left);
346       receive_return_value = 1;
347     } else {
348       sequence_number++;
349       received_file_bytes += receive_return_value;
```

```
350
351 #ifdef APPLICATION_LAYER_DEBUG_MODE
352       fprintf(stderr, "\treceive_return_value=%d\n",
              receive_return_value);
353       fprintf(stderr, "\treceived_file_bytes=%zu\n",
              received_file_bytes);
354 #endif
355
356       if ((fwrite(file_data, sizeof(char),
              receive_return_value,
357           received_file)) < 0) {
358         fprintf(stderr, "Error: file write error\n");
359         return -1;
360       }
361
362       if (receive_return_value > 0) {
363         free(file_data);
364       }
365     }
366   }
367
368 #ifdef APPLICATION_LAYER_DEBUG_MODE
369   fprintf(stderr,"receive_data_packets()\n");
370   fprintf(stderr,"\tfile_data_length=%zu\n",
          received_file_bytes);
371 #endif
372
373   if (attempts_left <= 0) {
374     return -1;
375   }
376
377   return fclose(received_file);
378 }
379
380 int parse_control_packet(const int control_packet_length,
      byte *control_packet,
381     char **file_name, size_t *file_size)
382 {
383 // TLV (file size)
384   if (control_packet[control_packet_t1_index]
385       != control_packet_tlv_type_filesize) {
386     fprintf(stderr, "parse_control_packet(): bad type 1");
387     return -1;
388   }
389   size_t v1_length = control_packet[control_packet_l1_index
          ];
390
391   if (v1_length != sizeof(size_t)) {
392     fprintf(stderr, "parse_control_packet(): bad L1 - file
          size length");
393     return -1;
394   }
395
396   size_t *file_size_tmp;
```

```
397
398   if ((file_size_tmp = malloc(sizeof(size_t))) == NULL ) {
399     perror("parse_control_packet() file_size_tmp malloc
          error");
400     return -1;
401   }
402
403   memcpy(file_size_tmp, (control_packet +
          control_packet_v1_index),
404          v1_length);
405   *file_size = *file_size_tmp;
406
407 // TLV (file name)
408   size_t control_packet_t2_index = control_packet_v1_index +
          v1_length + 1;
409   size_t control_packet_l2_index = control_packet_t2_index +
          1;
410   size_t control_packet_v2_index = control_packet_l2_index +
          1;
411
412   byte t2 = *(control_packet + control_packet_t2_index);
413
414   if (t2 != control_packet_tlv_type_name) {
415     fprintf(stderr, "parse_control_packet(): bad type 2");
416     free(file_size_tmp);
417     return -1;
418   }
419
420   size_t v2_length = *(control_packet +
          control_packet_l2_index);
421
422   if ((*file_name = malloc(v2_length * sizeof(char))) ==
          NULL ) {
423     perror("parse_control_packet() file_name malloc error");
424     free(file_size_tmp);
425     return -1;
426   }
427
428   memcpy(*file_name, (control_packet +
          control_packet_v2_index), v2_length);
429
430 #ifdef APPLICATION_LAYER_DEBUG_MODE
431   fprintf(stderr, "\tl1=%zu\n", v1_length);
432   fprintf(stderr, "\tfile_size=%zu\n", *file_size);
433   fprintf(stderr, "\tl2=%zu\n", v2_length);
434   fprintf(stderr, "\tname=%s\n", *file_name);
435 #endif
436   free(control_packet);
437   return 0;
438 }
439
440 int parse_data_packet(const int data_packet_length, byte *
      data_packet,
441     char **data, size_t* sequence_number)
```

```
442 {
443   int data_size = data_packet[data_packet_l2_index] * 256
444       + data_packet[data_packet_l1_index];
445 #ifdef APPLICATION_LAYER_DEBUG_MODE
446   fprintf(stderr, "parse_data_packet()\n");
447   fprintf(stderr, "\tcontrol_field=%d\n", data_packet[
          control_field_index]);
448   fprintf(stderr, "\tsequence_number=%d\n",
449       data_packet[data_packet_sequence_number_index]);
450   fprintf(stderr, "\tdata_size=%d\n", data_size);
451 #endif
452
453   if ((*data = malloc(sizeof(char) * data_size)) == NULL ) {
454     perror("parse_data_packet() data malloc error");
455     return -1;
456   }
457
458   memcpy(*data, (data_packet + data_packet_header_size *
          sizeof(byte)),
459       data_size);
460
461   size_t received_sequence_number =
462       data_packet[data_packet_sequence_number_index];
463   size_t expected_sequence_number = *sequence_number
464       % sequence_number_modulus;
465   if (received_sequence_number != expected_sequence_number)
          {
466 #ifdef APPLICATION_LAYER_DEBUG_MODE
467     fprintf(stderr, "bad packet sequence number: (received %
            zu) <-> (expected %zu)\n", received_sequence_number,
             expected_sequence_number);
468     free(data_packet);
469     return -1;
470 #endif
471     if (received_sequence_number > expected_sequence_number)
            {
472
473       while (*sequence_number % sequence_number_modulus
474           != received_sequence_number) {
475         (*sequence_number)++;
476         lost_packets++;
477       }
478     } else {
479       duplicated_packets++;
480       *sequence_number = expected_sequence_number;
481     }
482     //    free(data_packet);
483 //    free(*data);
484 //    return -1;
485   }
486   free(data_packet);
487   return data_size;
488 }
489
```

```
490 int llread(const int fd, byte **packet)
491 {
492   struct connection* c = &g_connections[fd];
493
494 // maximum size of a packet
495   size_t packet_size = c->packet_size * sizeof(byte);
496
497   if ((*packet = malloc(packet_size)) == NULL ) {
498     perror("llread() packet malloc error");
499     return -1;
500   }
501
502   int packet_length = 0;
503   if (c->is_active) {
504     if ((packet_length = receiver_read(c, *packet,
          packet_size,
505          NUM_FRAMES_PER_CALL)) < 0) {
506       fprintf(stderr, "llread(): error in receiver_read()\n"
           );
507       free(*packet);
508       return -1;
509     }
510   } else {
511     fprintf(stderr, "llread(): connection is not active\n");
512     free(*packet);
513     return -1;
514   }
515
516   return packet_length;
517 }
518
519 int receive_data_packet(const int fd, char **file_data,
520     size_t received_file_bytes, size_t* sequence_number)
521 {
522   byte *data_packet;
523   int data_packet_length = 0;
524
525   if ((data_packet_length = llread(fd, &data_packet)) < 0) {
526     return -1;
527   }
528
529 #ifdef APPLICATION_LAYER_DEBUG_MODE
530   fprintf(stderr, "receive_data_packet()\n");
531   fprintf(stderr, "\treceived_data_bytes=%d\n",
        data_packet_length);
532 #endif
533
534   byte control_field = data_packet[control_field_index];
535   if (control_field == control_field_data) {
536 #ifdef APPLICATION_LAYER_DEBUG_MODE
537     fprintf(stderr, "\tdata packet\n");
538 #endif
539
```

```
540       return parse_data_packet(data_packet_length, data_packet
              , file_data,
541           sequence_number);
542     } else if (control_field == control_field_end) {
543   #ifdef APPLICATION_LAYER_DEBUG_MODE
544       fprintf(stderr, "\tend control packet\n");
545   #endif
546       char* file_name;
547       size_t file_size;
548       if (parse_control_packet(data_packet_length, data_packet
              , &file_name,
549           &file_size) == 0) {
550         return 0;
551       } else {
552         return -1;
553       }
554     }
555
556     fprintf(stderr, "receive_data_packet(): bad control field
          value\n");
557     free(data_packet);
558     return -1;
559   }
560
561   int llopen(char *port, int transmitter)
562   {
563     struct connection conn;
564     strcpy(conn.port, port);
565     conn.frame_size = FRAME_SIZE;
566     conn.micro_timeout_ds = MICRO_TIMEOUT_DS;
567     conn.timeout_s = TIMEOUT_S;
568     conn.num_retransmissions = NUM_RETRANSMISSIONS;
569     conn.close_wait_time = CLOSE_WAIT_TIME;
570     conn.packet_size = FRAME_SIZE;
571
572     if ((conn.is_transmitter = transmitter)) {
573       if (transmitter_connect(&conn) < 0) {
574         return -1;
575       }
576     } else {
577       if (receiver_listen(&conn)) {
578         return -1;
579       }
580     }
581
582     g_connections[conn.fd] = conn;
583     return conn.fd;
584   }
585
586   void print_status(time_t t0, size_t num_bytes, unsigned long
          counter)
587   {
588     double dt = difftime(time(NULL ), t0);
589     double speed = ((double) (num_bytes * 8)) / dt;
```

```
590   fprintf(stderr, "----------------------\n");
591   fprintf(stderr,
592       "Link layer transmission %ld: %lf bit per sec; %ldB of
          data\n",
593       counter, speed, num_bytes);
594   fprintf(stderr, "----------------------\n");
595 }
596
597 int llclose(const int fd)
598 {
599   return disconnect(&g_connections[fd]);
600 }
601
602 void receiver_stats()
603 {
604   fprintf(stdout, "Receiver statistics\n");
605   fprintf(stdout, "\treceived file bytes/file bytes:%zu/%zu\
      n",
606       received_file_bytes, real_file_bytes);
607   fprintf(stdout, "\tlost packets:%zu\n", lost_packets);
608   fprintf(stdout, "\tduplicated packets:%zu\n",
      duplicated_packets);
609 }
610
611 void retry(int* attempt)
612 {
613   (*attempt)--;
614   if (*attempt <= 0)
615     return;
616 #ifdef APPLICATION_LAYER_DEBUG_MODE
617   fprintf(stderr, "\tnew attempt in 5 seconds .");
618 #endif
619   sleep(1);
620 #ifdef APPLICATION_LAYER_DEBUG_MODE
621   fprintf(stderr, " .");
622 #endif
623   sleep(1);
624 #ifdef APPLICATION_LAYER_DEBUG_MODE
625   fprintf(stderr, " .");
626 #endif
627   sleep(1);
628 #ifdef APPLICATION_LAYER_DEBUG_MODE
629   fprintf(stderr, " .");
630 #endif
631   sleep(1);
632 #ifdef APPLICATION_LAYER_DEBUG_MODE
633   fprintf(stderr, " .\n");
634 #endif
635   sleep(1);
636 }
```

A.2 Camada de ligação de dados

serial_port.c

```
1  #include <sys/types.h>
2  #include <sys/stat.h>
3  #include <fcntl.h>
4  #include <termios.h>
5  #include <stdio.h>
6  #include <strings.h>
7  #include <stdlib.h>
8  #include <unistd.h>
9  #include <fcntl.h>
10 #include <errno.h>
11
12 #include "netlink.h"
13 #include "serial_port.h"
14
15 typedef unsigned char byte;
16 int serial_port_baudrate = B19200;
17
18 #define _POSIX_SOURCE 1 /* POSIX compliant source */
19
20 struct termios g_oldtio;
21
22 byte g_previous_last_byte = 0;
23 byte serial_port_previous_last_byte()
24 {
25   return g_previous_last_byte;
26 }
27
28 byte g_last_byte = 0;
29 byte serial_port_last_byte()
30 {
31   return g_last_byte;
32 }
33
34 int serial_port_open(const char *dev_name, const int
       micro_timeout)
35 {
36 #ifdef SERIAL_PORT_DEBUG_MODE
37   fprintf(stderr,"serial_port_open(): entering function; dev
         = %s\n, timeout = \
38              %d\n",dev_name,micro_timeout);
39 #endif
40
41   /*
42    Open serial port device for reading and writing and not
         as controlling
43    tty because we don't want to get killed if linenoise
         sends CTRL-C.
44    */
45   int fd = -1;
46   struct termios newtio;
```

```
47
48    fd = open(dev_name, O_RDWR | O_NOCTTY);
49    if (fd < 0) {
50      perror(dev_name);
51      return -1;
52    }
53
54  #ifdef SERIAL_PORT_DEBUG_MODE
55    fprintf(stderr,"isatty()=%d, ttyname()=%s\n",isatty(fd),
          ttyname(fd));
56    fprintf(stderr,"fd = %d\n",fd);
57  #endif
58
59    /* Port settings */
60    if (tcgetattr(fd, &g_oldtio) == -1) { /* save current port
          settings */
61      perror("tcgetattr");
62      return -1;
63    }
64
65    bzero(&newtio, sizeof(newtio)); /* clear struct for new
          port settings */
66    newtio.c_cflag = serial_port_baudrate | CS8 | CLOCAL |
          CREAD;
67    newtio.c_iflag = IGNPAR;
68    newtio.c_oflag = 0;
69    newtio.c_lflag = 0;
70
71    /* 0 => inter-character timer unused */
72    newtio.c_cc[VTIME] = micro_timeout;
73
74    /* VMIN=1 => blocking read until 1 character is received
          */
75    //newtio.c_cc[VMIN] = (micro_timeout == 0) ? 1 : 0;
76    newtio.c_cc[VMIN] = 1;
77
78  #ifdef SERIAL_PORT_DEBUG_MODE
79    fprintf(stderr,"serial_port_open(): timeout=%d, c_cc[VMIN
          ]=%d\n",micro_timeout,
80        newtio.c_cc[VMIN]);
81  #endif
82
83    tcflush(fd, TCIFLUSH);
84    if (tcsetattr(fd, TCSANOW, &newtio) == -1) {
85      perror("tcsetattr");
86      exit(-1);
87    }
88    return fd;
89  }
90
91  int serial_port_close(int fd, int close_wait_time)
92  {
93  #ifdef SERIAL_PORT_DEBUG_MODE
```

```
94    fprintf(stderr,"serial_port_close(): waiting %d seconds to
          close...\n",
95          close_wait_time);
96  #endif
97    sleep(close_wait_time);
98
99    int ret = tcsetattr(fd, TCSANOW, &g_oldtio);
100   if (ret == -1) {
101     perror("tcsetattr");
102     return -1;
103   }
104   return close(fd);
105 }
106
107 int serial_port_write(int fd, byte *data, int len)
108 {
109 #ifdef SERIAL_PORT_DEBUG_MODE
110   fprintf(stderr,"serial_port_write(): writting: length = %d
          , fd = %d\n",len,fd);
111 #endif
112
113     //fprintf(stderr,"write\n");
114   int result = write(fd, data, len);
115
116   if (result < 0) {
117     fprintf(stderr, "serial_port_write(): error %d, errno =
          %x\n", result,
118         errno);
119 #ifdef SERIAL_PORT_DEBUG_MODE
120   } else {
121     fprintf(stderr,"serial_port_write(): wrote %d bytes\n",
          result);
122 #endif
123   }
124
125   return result;
126 }
127
128 /** \brief Read from serial port until either:
129  * - a delimiter char is found
130  * - the maximum number of chars is read
131  * - there is a timeout
132  *
133  * @return Number of chars read or negative number if error
134  * */
135 int serial_port_read(int fd, byte *data, byte delim, int
      maxc)
136 {
137 #ifdef SERIAL_PORT_DEBUG_MODE
138   fprintf(stderr,"serial_port_read(): entering function\n");
139   fprintf(stderr,"                    delimiter = %x\n",(
        char)delim);
140 #endif
141
```

```
142   g_previous_last_byte = g_last_byte;
143
144     //fprintf(stderr,"read\n");
145   byte *p = data;
146   int nc = 0; // num chars read so far
147   do {
148     int ret = read(fd, &g_last_byte, 1);
149     if (ret == 0) {
150 #ifdef SERIAL_PORT_DEBUG_MODE
151       fprintf(stderr,"serial_port_read(): micro timeout tick
            \n");
152 #endif
153       break;
154     } else if (ret < 0 && errno == EINTR) { // interrupted,
          possibly by an alarm
155 #ifdef SERIAL_PORT_DEBUG_MODE
156     fprintf(stderr,"serial_port_read(): received interrupt\n
          ");
157 #endif
158       return 0;
159     } else if (ret < 0) {
160 #ifdef SERIAL_PORT_DEBUG_MODE
161       fprintf(stderr,"serial_port_read(): ret = %d, errno =
            %d\n",ret,errno);
162 #endif
163     }
164 #ifdef SERIAL_PORT_PRINT_ALL_CHARS
165     fprintf(stderr,"< %x\n",g_last_byte);
166     //fprintf(stderr,"c: %d/%c, nc: %d, ret: %d\n",c,c,nc,
          ret);
167 #endif
168
169     *p++ = g_last_byte;
170     nc++;
171   } while (nc < maxc && g_last_byte != delim);
172
173 #ifdef SERIAL_PORT_DEBUG_MODE
174   //fprintf(stderr,"serial_port_read(): read (%d): %.*s\n",
        nc,nc,data);
175 #endif
176
177   return nc;
178 }
```

data_link.c

```
1  #include <stdio.h>
2
3  #include "serial_port.h"
4  #include "data_link.h"
5  #include "data_link_codes.h"
6  #include "data_link_io.h"
7  #include <string.h>
8  #include <stdlib.h>
```

```
9
10 #define TRUE 1
11 #define FALSE 0
12
13 static long g_use_limited_rejected_retries = 1; // true or
       false
14
15 byte data_reply_byte(unsigned long frame_number, int
       accepted)
16 {
17   return (accepted ? C_RR : C_REJ) | ((frame_number % 2) ? 0
         : (1 << 7));
18 }
19
20 byte data_control_byte(unsigned long frame_number)
21 {
22   return (frame_number % 2 == 0) ? 0 : (1 << 6);
23 }
24
25 static int handle_disconnect(struct connection* conn)
26 {
27   int ret = 0;
28
29   int ntries = conn->num_retransmissions;
30   while (1) {
31     struct frame reply;
32     if ((ntries = f_send_acknowledged_frame(conn->fd, ntries
           ,
33          conn->timeout_s, DISC, &reply)) < 0) {
34       ret = -1;
35       break;
36     }
37     if (reply.control == C_UA) {
38       break;
39     }
40   }
41
42   conn->is_active = 0;
43   if (serial_port_close(conn->fd, 0) < 0 || ret < 0) {
44     return -1;
45   }
46   return 0;
47 }
48
49 /*
50  * TRANSMITTER
51  */
52
53 /** \brief Establish logical connection.
54  *
55  * Open serial port, send SET, receive UA. */
56 int transmitter_connect(struct connection* conn)
57 {
58   conn->is_active = 0;
```

```
59    conn->max_buffer_size = LL_MAX_PAYLOAD_STUFFED;
60    conn->frame_number = 0;
61
62    if ((conn->fd = serial_port_open(conn->port, conn->
          micro_timeout_ds)) < 0) {
63  #ifdef DATA_LINK_DEBUG_MODE
64      fprintf(stderr,
65          "transmitter_connect(): could not open %s\n",conn->
              port);
66  #endif
67      return conn->fd;
68    }
69
70    /* Send SET frame and receive UA.  */
71    int ntries = conn->num_retransmissions;
72    while (1) {
73      struct frame reply;
74      if ((ntries = f_send_acknowledged_frame(conn->fd, ntries
          ,
75          conn->timeout_s, SET, &reply)) < 0) {
76  #ifdef DATA_LINK_DEBUG_MODE
77        fprintf(stderr,"transmitter_connect(): no connection.\
              n");
78  #endif
79        return -1;
80      }
81
82      if (reply.control == C_UA) {
83        break;
84      }
85    }
86
87    conn->is_active = 1;
88  #ifdef DATA_LINK_DEBUG_MODE
89    fprintf(stderr,"Connection established.\n");
90  #endif
91    return 0;
92  }
93
94  int transmitter_write(struct connection* conn, byte*
        data_packet, size_t size)
95  {
96    fprintf(stderr," #-################################### \n
          ");
97    fprintf(stderr,"\n");
98    fprintf(stderr," BEGIN TRANSMIT %zu\n", conn->frame_number
          );
99    fprintf(stderr,"\n");
100   fprintf(stderr," ##################################### \n
          ");
101
102   struct frame out_frame = { .address = A, .control =
          data_control_byte(
```

```
103         conn->frame_number), .size = size, .data = data_packet
                };
104
105     byte success_rep = data_reply_byte(conn->frame_number,
            TRUE);
106     byte rej_rep = data_reply_byte(conn->frame_number, FALSE);
107       fprintf(stderr,"success_rep = %x\n",success_rep);
108       fprintf(stderr,"rej_rep = %x\n",rej_rep);
109
110     /* Send data frame and receive confirmation.  */
111     int ntries = conn->num_retransmissions;
112     while (1) {
113       struct frame reply_frame;
114           fprintf(stderr,"trying to send frame %lu\n",conn->
                  frame_number);
115       if ((ntries = f_send_acknowledged_frame(conn->fd, ntries
              ,
116           conn->timeout_s, out_frame, &reply_frame)) < 0) {
117               fprintf(stderr,"failed acknowledged frame\n");
118         return -1;
119       }
120           fprintf(stderr," ---- control = %x\n",reply_frame.
                  control);
121       if (reply_frame.control == rej_rep) {
122               fprintf(stderr,"rejected frame\n");
123         if (g_use_limited_rejected_retries) {
124           --ntries;
125         }
126       }
127       if (reply_frame.control == success_rep) {
128               fprintf(stderr,"accepted frame\n");
129         break;
130       }
131     }
132
133     conn->frame_number++;
134       fprintf(stderr,"new frame number: %zu\n",conn->
              frame_number);
135     return 0;
136 }
137
138 /** \brief Establish logical connection.
139  *
140  * Open serial port, send SET, receive UA. */
141 int disconnect(struct connection* conn)
142 {
143     int return_value = 0;
144
145     /* Send DISC and receive DISC.  */
146     int ntries = conn->num_retransmissions;
147     while (1) {
148       struct frame reply;
149       if ((ntries = f_send_acknowledged_frame(conn->fd, ntries
              ,
```

```
150           conn->timeout_s, DISC, &reply)) < 0) {
151         return_value = -1;
152         break;
153       }
154       if (reply.control == C_DISC) {
155         break;
156       }
157     }
158
159     if (return_value >= 0) {
160       if (f_send_frame(conn->fd, UA) != SUCCESS_CODE) {
161         return_value = -1;
162       }
163     }
164
165     conn->is_active = 0;
166
167     // Close port
168     if (serial_port_close(conn->fd, conn->close_wait_time) <
          0) {
169       return_value = -1;
170     }
171
172   #ifdef DATA_LINK_DEBUG_MODE
173     if (return_value < 0) {
174       fprintf(stderr,"disconnect(): failed to close connection
            .\n");
175     }
176   #endif
177     return return_value;
178   }
179
180   /*
181    * RECEIVER
182    */
183
184   // TODO
185   int receiver_listen(struct connection* conn)
186   {
187     conn->max_buffer_size = LL_MAX_PAYLOAD_STUFFED;
188     conn->frame_number = 0;
189
190     if ((conn->fd = serial_port_open(conn->port, conn->
          micro_timeout_ds)) < 0) {
191   #ifdef DATA_LINK_DEBUG_MODE
192       fprintf(stderr,"listen(): could not open %s\n",conn->
            port);
193   #endif
194       return conn->fd;
195     }
196
197     while (1) {
198       struct frame in;
199       if (f_receive_frame(conn->fd, &in, 0) == ERROR_CODE) {
```

```
200         return -1;
201       }
202           fprintf(stderr,"receiver_listen: %x\n",in.control);
203       if (in.control == C_SET) {
204         f_send_frame(conn->fd, UA);
205         conn->is_active = 1;
206 #ifdef DATA_LINK_DEBUG_MODE
207         fprintf(stderr,"listen(): connection established.\n");
208 #endif
209         return 0;
210       }
211     }
212 }
213
214 int receiver_read(struct connection* conn, byte *begin,
        size_t max_data_size,
215       const int max_num_frames)
216 {
217   int num_frames = 0;
218   byte *p = begin;
219   byte *end = begin + max_data_size;
220
221   while (p < end && (num_frames < max_num_frames ||
          max_num_frames == 0)) {
222     fprintf(stderr," ######################################
          \n");
223     fprintf(stderr,"\n");
224     fprintf(stderr," BEGIN RECEIVE %zu\n", conn->
          frame_number);
225     fprintf(stderr,"\n");
226     fprintf(stderr," ######################################
          \n");
227
228     struct frame in;
229     in.data = p;
230     in.max_data_size = end - p;
231     Return_e ret = f_receive_frame(conn->fd, &in, 0);
232
233     if (ret == ERROR_CODE) {
234       return -1;
235     } else if (ret == TIMEOUT_CODE) {
236       break;
237     } else if (ret == BADFRAME_CODE) {
238 #ifdef DATA_LINK_DEBUG_MODE
239       fprintf(stderr,"receiver_read(): parsing: bad frame.\n
            ");
240 #endif
241       /*
242        * Send 'bad frame' acknowledgment.
243        */
244       byte c_out = data_reply_byte(conn->frame_number, FALSE
            );
245       if (f_send_frame(conn->fd, FRAME(c_out)) !=
            SUCCESS_CODE) {
```

```
246           break;
247         }
248       } else if (in.control == C_DISC) {
249         handle_disconnect(conn);
250         break;
251       } else if (in.control != data_control_byte(conn->
              frame_number)) {
252         /*
253          * Frame number mismatch.
254          */
255 #ifdef DATA_LINK_DEBUG_MODE
256         fprintf(stderr,"receiver_read(): bad control byte.\n")
                ;
257         fprintf(stderr,"receiver_read(): %x, %x.\n",
258             in.control,data_control_byte(conn->frame_number));
259 #endif
260         byte control = data_reply_byte(conn->frame_number,
                FALSE);
261         // reject this frame
262         if (f_send_frame(conn->fd, FRAME(control)) !=
                SUCCESS_CODE) {
263           break;
264         }
265       } else {
266 #ifdef DATA_LINK_DEBUG_MODE
267         fprintf(stderr,"receiver_read(): received: \"%.*s\".\n
                ",
268             (int)in.size,p);
269 #endif
270
271         num_frames++;
272         p += in.size;
273
274         byte control = data_reply_byte(conn->frame_number,
                TRUE);
275         if (f_send_frame(conn->fd, FRAME(control)) !=
                SUCCESS_CODE) {
276           break;
277         }
278         conn->frame_number++;
279               fprintf(stderr,"new frame number: %zu\n",conn->
                    frame_number);
280       }
281     }
282     return p - begin;
283 }
284
285 // TODO
286 int wait_for_disconnect(struct connection* conn, int timeout
        )
287 {
288     while (1) {
289       struct frame in;
290       f_receive_frame(conn->fd, &in, 0);
```

```
291       if (in.control == C_DISC) {
292         return handle_disconnect(conn);
293       } else {
294 #ifdef DATA_LINK_DEBUG_MODE
295         fprintf(stderr,
296             "receiver_wait_disconnect(): frame ignored, C=%x.\
                n",
297             in.control);
298 #endif
299       }
300     }
301     return -1;
302 }
303
304 #if 0
305 // ------------------------
306
307 // Function to print the pack content
308 char* packet_content(const char* packet, const int size)
309 {
310     const char *hex = "0123456789ABCDEF";
311     char *content = (char *) malloc(sizeof(char) * (3 * size))
          ;
312     char *pout = content;
313     const char *pin = packet;
314
315     int i;
316
317     if (pout) {
318
319       for (i = 0; i < size - 1; i++) {
320         *pout++ = hex[(*pin>>4)&0xF];
321         *pout++ = hex[(*pin++)&0xF];
322         *pout++ = ':';
323       }
324       *pout++ = hex[(*pin>>4)&0xF];
325       *pout++ = hex[(*pin++)&0xF];
326       *pout = 0;
327     }
328
329     return content;
330 }
331 #endif
```

data_link_io.c

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <signal.h>
4 #include <stdlib.h>
5
6 #include "serial_port.h"
7 #include "data_link.h"
8 #include "data_link_io.h"
```

```
9  #include "data_link_codes.h"
10 #include "byte.h"
11
12 volatile int g_timeout_alarm = 0;
13 void set_timeout_alarm()
14 {
15   g_timeout_alarm = 1;
16 }
17
18 static byte g_buffer[LL_MAX_FRAME_SZ]; /** Local array for
       frame building. */
19 static long g_sent_frame_counter = 0;
20 static long g_rec_frame_counter = 0;
21 static long g_header_bcc_error_counter = 0;
22 static long g_data_bcc_error_counter = 0;
23
24 struct frame FRAME(const byte control)
25 {
26   struct frame super = { .address = A, .control = control, .
       size = 0 };
27   return super;
28 }
29
30 void f_print_frame(const struct frame frame)
31 {
32   fprintf(stderr, "Frame:\n");
33   fprintf(stderr, "A:%o C:%o S:%zu\n", frame.address, frame.
       control,
34           frame.size);
35   if (frame.size > 0) {
36     for (int i = 0; i < frame.size; i++) {
37       putc(frame.data[i], stderr);
38     }
39     putc('\n', stderr);
40   }
41   putc('\n', stderr);
42 }
43
44 void f_dump_frame_buffer(const char *filename)
45 {
46   FILE* f;
47   if ((f = fopen(filename, "w")) == NULL ) {
48     fprintf(stderr, "f_dump_frame_buffer(): file error: line
         : %d\n",
49             __LINE__);
50   } else if (fprintf(f, "%.*s", LL_MAX_FRAME_SZ, g_buffer) <
       0) {
51     fprintf(stderr, "f_dump_frame_buffer(): file error: line
         : %d\n",
52             __LINE__);
53   } else if (fclose(f) == EOF) {
54     fprintf(stderr, "f_dump_frame_buffer(): file error: line
         : %d\n",
55             __LINE__);
```

```
56    }
57  }
58
59  /* Reads array and builds a "struct Frame* frame*" from it
60   * - checks if its a supervision or data frame
61   * - checks bcc
62   * - destuffs bytes
63  netlink.c * - returns SUCCESS_CODE, ERROR_CODE, or
        BADFRAME_CODE (when bcc is wrong, or
64   *   data is too large)
65   */
66  static Return_e parse_frame_from_array(struct frame* frame,
        byte *a)
67  {
68  #ifdef DATA_LINK_DEBUG_MODE
69    fprintf(stderr,"  parse_frame_from_array(): entering
          function.\n");
70    //f_dump_frame_buffer("FRAME");
71  #endif
72
73    if (*a++ != FLAG) {
74  #ifdef DATA_LINK_DEBUG_MODE
75      fprintf(stderr,"  parse_frame_from_array(): error:
            missing flag: \
76                  (line %d).\n",__LINE__);
77  #endif
78      return ERROR_CODE;
79    }
80    for (int i = 0; i <= 2; i++) { // the next fields should
          not have a FLAG
81      if (a[i] == FLAG) {
82  #ifdef DATA_LINK_DEBUG_MODE
83        fprintf(stderr,"  parse_frame_from_array(): error:
              unexpected flag \
84                    (line %d).\n",__LINE__);
85  #endif
86        return BADFRAME_CODE;
87      }
88    }
89
90    frame->address = *a++;
91    frame->control = *a++;
92    const byte header_bcc = *a++;
93    if (header_bcc != (frame->address ^ frame->control)) {
94  #ifdef DATA_LINK_DEBUG_MODE
95      fprintf(stderr,"  parse_frame_from_array(): error:
            header bcc: %d.\n",__LINE__);
96  #endif
97      ++g_header_bcc_error_counter;
98      return BADFRAME_CODE;
99    }
100   frame->size = 0;
101
102   /* Supervision frame */
```

```
103   if (*a == FLAG) {
104 #ifdef DATA_LINK_DEBUG_MODE
105     fprintf(stderr,"  parse_frame_from_array(): read a
          supervision frame.\n");
106 #endif
107     return SUCCESS_CODE;
108   }
109
110   if (*(a + 1) == FLAG) {
111 #ifdef DATA_LINK_DEBUG_MODE
112     fprintf(stderr,"  parse_frame_from_array(): error: only
          1B remaining.\n");
113 #endif
114     return BADFRAME_CODE;
115   }
116
117   /* Data frame */
118   byte data_bcc = 0;
119   size_t num_bytes = 0;
120   while (1) {
121     if (num_bytes > LL_MAX_PAYLOAD_STUFFED) {
122 #ifdef DATA_LINK_DEBUG_MODE
123       fprintf(stderr,"  parse_frame_from_array(): line %d.\n
            ",__LINE__);
124 #endif
125       return BADFRAME_CODE;
126     }
127     if (frame->size > frame->max_data_size) {
128 #ifdef DATA_LINK_DEBUG_MODE
129       fprintf(stderr,"  parse_frame_from_array(): line %d.\n
            ",__LINE__);
130 #endif
131       return BADFRAME_CODE;
132     }
133
134     byte c;
135     if (a[num_bytes] == BS_ESC) {
136       fprintf(stderr, "----\n");
137       // remove byte stuffing
138       ++num_bytes;
139       c = BS_OCT ^ a[num_bytes];
140     } else {
141       c = a[num_bytes];
142     }
143         ++num_bytes;
144         frame->data[frame->size++] = c;
145         fprintf(stderr, "%x\n", c);
146
147     /*
148      * Stop loop condition.
149      */
150     if (a[num_bytes] == FLAG) {
151       if (frame->data[frame->size-1] != data_bcc) {
152         ++g_data_bcc_error_counter;
```

```
153 #ifdef DATA_LINK_DEBUG_MODE
154         fprintf(stderr,"parse_frame_from_array(): data bcc
              error: line %d.\n",__LINE__);
155         fprintf(stderr, "frame size %ld\n", frame->size);
156         fprintf(stderr, "data bcc = %x\n", data_bcc);
157         fprintf(stderr, "a[num_bytes-1] = %x\n", a[num_bytes
              -1]);
158 #endif
159         return BADFRAME_CODE;
160       }
161 #ifdef DATA_LINK_DEBUG_MODE
162       fprintf(stderr,"  parse_frame_from_array(): successful
            read.\n");
163 #endif
164       return SUCCESS_CODE;
165     } else {
166           data_bcc ^= c;
167           fprintf(stderr,"c = %x, bcc = %x\n",c,data_bcc);
168         }
169
170   }
171 }
172
173 static byte *
174 copy_and_stuff_bytes(byte *dest, const byte *src, const
      size_t src_size)
175 {
176   int bcc = 0;
177   for (int i = 0; i <= src_size; ++i) {
178         byte c;
179         if (i != src_size) {
180           c = src[i];
181           bcc ^= c;
182           fprintf(stderr,"c = %x, bcc = %x\n",c,bcc);
183           //fprintf(stderr, "%2x\n", c);
184         } else {
185           fprintf(stderr,"bcc = %x\n",bcc);
186           c = bcc;
187         }
188     if (c == FLAG || c == BS_ESC) {
189       *dest++ = BS_ESC;
190       *dest++ = BS_OCT ^ c;
191     } else {
192       *dest++ = c;
193     }
194   }
195   fprintf(stderr, "size %ld (not counting bcc)\n", src_size)
        ;
196   return dest;
197 }
198
199 /** \brief Send any type of frame.
200  *
```

```
201   * Compose a g_buffer[] array from a Frame and send it to
          the serial port.
202   *
203   * @return ERROR_CODE or ERROR_SUCCESS.
204   */
205  Return_e f_send_frame(const int fd, const struct frame frame
          )
206  {
207  #ifdef DATA_LINK_DEBUG_MODE
208    fprintf(stderr,"f_send_frame(): beginning frame writing (C
          =0x%2x, %zu bytes)\n",
209         frame.control,frame.size);
210  #endif
211
212    byte *bp = g_buffer;
213
214    *bp++ = FLAG;
215
216    // write header
217    *bp++ = frame.address;
218    *bp++ = frame.control;
219    *bp++ = frame.address ^ frame.control; // bcc
220
221    // copy data
222    if (frame.size > LL_MAX_PAYLOAD_UNSTUFFED) {
223  #ifdef DATA_LINK_DEBUG_MODE
224      fprintf(stderr,"f_send_frame(): tried to send too big a
             frame \
225               (%zu bytes)\n",frame.size);
226  #endif
227      return ERROR_CODE;
228
229    } else if (frame.size > 0) { // frame might be 0 if it is
          supervision
230      bp = copy_and_stuff_bytes(bp, frame.data, frame.size);
231  #ifdef DATA_LINK_DEBUG_MODE
232      fprintf(stderr,"f_send_frame(): unstuffed data size %ld
             .\n",frame.size);
233  #endif
234    }
235
236    *bp++ = FLAG;
237
238    if (serial_port_write(fd, g_buffer, bp - g_buffer) < 0) {
239  #ifdef DATA_LINK_DEBUG_MODE
240      fprintf(stderr,"f_send_frame(): writting failed.\n");
241  #endif
242      return ERROR_CODE;
243    }
244    ++g_sent_frame_counter;
245  #ifdef DATA_LINK_DEBUG_MODE
246    fprintf(stderr,"f_send_frame(): finished sending frame #%
          ld\n",
247         g_sent_frame_counter);
```

```
248 #endif
249   return SUCCESS_CODE;
250 }
251
252 void start_alarm(int s)
253 {
254 #ifdef DATA_LINK_DEBUG_MODE
255   fprintf(stderr,"Setting alarm: %d sec.\n",s);
256 #endif
257   signal(SIGALRM, set_timeout_alarm);  // TODO: put in init
        function
258   g_timeout_alarm = 0;
259   alarm(s);
260 }
261
262 /**
263  */
264 Return_e f_receive_frame(const int fd, struct frame* frame,
      const int timeout_s)
265 {
266 #ifdef DATA_LINK_DEBUG_MODE
267   fprintf(stderr,"  f_receive_frame(): beginning frame
        reception.\n");
268 #endif
269
270   const int using_timeout = (timeout_s > 0);
271
272   if (using_timeout) {
273     start_alarm(timeout_s);
274   }
275   while (1) {
276     while (serial_port_last_byte() != FLAG) { // first flag
277 #ifdef DATA_LINK_DEBUG_MODE
278     fprintf(stderr,"  f_receive_frame(): looking for next
          flag.\n");
279 #endif
280
281       if (serial_port_read(fd, g_buffer, FLAG,
            LL_MAX_FRAME_SZ) < 0) {
282         return ERROR_CODE;
283       }
284       if (using_timeout && g_timeout_alarm) {
285         return TIMEOUT_CODE;
286       }
287
288 #ifdef DATA_LINK_DEBUG_MODE
289       fprintf(stderr,"  f_receive_frame(): last byte=%x.\n",
290           serial_port_last_byte());
291 #endif
292     }
293 #ifdef DATA_LINK_DEBUG_MODE
294     fprintf(stderr,"  f_receive_frame(): First FLAG detected
          .\n");
295 #endif
```

```
296       g_buffer[0] = FLAG;
297
298       // skip initial flags and read
299       while (1) {
300         if (using_timeout) {
301           start_alarm(timeout_s);
302         }
303         int ret = serial_port_read(fd, g_buffer + 1, FLAG,
304             LL_MAX_FRAME_SZ - 1);
305         if (using_timeout && g_timeout_alarm) {
306           return TIMEOUT_CODE;
307         }
308         if (ret < 0) {
309           fprintf(stderr, "  f_receive_frame(): error.\n");
310           return -1;
311         }
312         if (ret > 1) {
313           break;
314         }
315       }
316
317       if (serial_port_last_byte() == FLAG) { // final flag
318   #ifdef DATA_LINK_DEBUG_MODE
319       fprintf(stderr,"  f_receive_frame(): Last FLAG detected
              .\n");
320   #endif
321         Return_e ret = parse_frame_from_array(frame, g_buffer)
                ;
322         if (ret == SUCCESS_CODE) {
323           ++g_rec_frame_counter;
324         }
325               else if (ret == BADFRAME_CODE) {
326                   fprintf(stderr,"bad frame detected while
                          parsing\n");
327               }
328         return ret;
329       }
330       if (using_timeout && g_timeout_alarm) {
331         return TIMEOUT_CODE;
332       }
333     }
334   }
335
336   /** Sends 'frame' and gets reply. */
337   int f_send_acknowledged_frame(const int fd, const unsigned
          num_retransmissions,
338       const int timeout_s, struct frame out_frame, struct
              frame *reply)
339   {
340     int ntries = (num_retransmissions <= 0) ? -1 :
            num_retransmissions;
341
342     while (ntries > 0) {
343   #ifdef DATA_LINK_DEBUG_MODE
```

```
344     fprintf(stderr,"f_send_acknowledged_frame(): ntries = %d
            .\n",ntries);
345 #endif
346
347     if (f_send_frame(fd, out_frame) == ERROR_CODE) {
348 #ifdef DATA_LINK_DEBUG_MODE
349       fprintf(stderr,"f_send_acknowledged_frame(): error
              writting\n");
350 #endif
351       return -1;
352     }
353
354     reply->control = 0;
355     Return_e ret = f_receive_frame(fd, reply, timeout_s);
356
357     if (ret == ERROR_CODE) {
358 #ifdef DATA_LINK_DEBUG_MODE
359       fprintf(stderr,"f_send_acknowledged_frame(): error
              reading\n");
360 #endif
361       return -1;
362     } else if (ret == TIMEOUT_CODE) {
363 #ifdef DATA_LINK_DEBUG_MODE
364       fprintf(stderr,"f_send_acknowledged_frame(): timeout\n
              ");
365 #endif
366       --ntries;
367       continue;
368     } else if (ret == BADFRAME_CODE) {     // reset number
            of attempts
369 #ifdef DATA_LINK_DEBUG_MODE
370         fprintf(stderr,"f_send_acknowledged_frame(): bad
              frame\n");
371 #endif
372       ntries--;
373       continue;
374         } else if ((reply->control & 7) == C_REJ) {
375             fprintf(stderr,"detected rejected frame while
                  parsing\n");
376             ntries--;
377     } else {
378       break;
379     }
380   }
381
382   return ntries;
383 }
```

A.3 Ficheiros com funções de validação

data_link_test.c

```
1 #include "data_link_codes.h"
2 #include "data_link_io.h"
```

```c
#include "data_link.h"
#include "serial_port.h"
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include <unistd.h>
#include <termios.h> // Baudrate
#define DEVICE "/dev/ttyS0"

int TRANSMITTER = 0;

struct connection CONNECTION = { .max_buffer_size = LL_MAX_PAYLOAD_STUFFED,
    .num_retransmissions = 3, .baudrate = B300, .timeout_s = 3,
    .micro_timeout_ds = 11, .close_wait_time = 3 };

int are_frames_equal(struct frame f1, struct frame f2)
{
  if (f1.address != f2.address) {
    fprintf(stderr, "are_frames_equal(): %d\n", __LINE__);
    return 0;
  }
  if (f1.control != f2.control) {
    fprintf(stderr, "are_frames_equal(): %d\n", __LINE__);
    return 0;
  }
  if (f1.size != f2.size) {
    fprintf(stderr, "are_frames_equal(): %d\n", __LINE__);
    return 0;
  }
  for (size_t i = 0; i < f1.size; i++) {
    if (f1.data[i] != f2.data[i]) {
      fprintf(stderr, "are_frames_equal(): %d\n", __LINE__);
      return 0;
    }
  }
  return 1;
}

// Print how the arguments must be
void help(char **argv)
{
  printf("Usage: %s [OPTION] <serial port>\n", argv[0]);
  printf("\n Program options:\n");
  printf("  -t          transmit data over the serial port\n");
}

// Verifies serial port argument
int parse_serial_port_arg(int index, char **argv)
{
  char *dev = argv[index];
```

```
53    if ((strcmp("/dev/ttyS0", dev) != 0) && (strcmp("/dev/
          ttyS1", dev) != 0)
54        && (strcmp("/dev/ttyS4", dev) != 0)) {
55      return -3;
56    }
57    return index;
58  }
59
60  // Verifies arguments
61  int parse_args(int argc, char **argv)
62  {
63    if (argc < 2)
64      return -1;
65
66    if (argc == 2)
67      return parse_serial_port_arg(1, argv);
68
69    if (argc == 3) {
70      if ((strcmp("-t", argv[1]) != 0))
71        return -2;
72      else
73        TRANSMITTER = 1;
74
75      return parse_serial_port_arg(2, argv);
76    } else
77      return -1;
78  }
79
80  int test_1(struct connection* conn)
81  {
82    if (TRANSMITTER) {
83      f_print_frame(SET);
84      if (f_send_frame(conn->fd, SET) != SUCCESS_CODE) {
85        f_dump_frame_buffer("FRAME");
86        printf("line: %d\n", __LINE__);
87        return 1;
88      }
89
90    } else {
91
92      struct frame frame;
93      Return_e ret = f_receive_frame(conn->fd, &frame, 0);
94      f_print_frame(frame);
95      f_print_frame(SET);
96      f_dump_frame_buffer("FRAME");
97
98      if (ret != SUCCESS_CODE) {
99        printf("ret: %d\n", (int) ret);
100       printf("line: %d\n", __LINE__);
101       return 1;
102     }
103     if (!are_frames_equal(frame, SET)) {
104       printf("line: %d\n", __LINE__);
105       return 1;
```

```
106       }
107     }
108     return 0;
109   }
110
111   int test_2(struct connection* conn)
112   {
113     if (TRANSMITTER) {
114       sleep(1);
115
116       f_print_frame(SET);
117       struct frame reply;
118       if (0 > f_send_acknowledged_frame(conn->fd, 1, 10, SET,
              &reply)) {
119         printf("line: %d\n", __LINE__);
120         return -1;
121       }
122       if (reply.control != C_UA) {
123         f_dump_frame_buffer("FRAME");
124         printf("line: %d\n", __LINE__);
125         return -1;
126       }
127     } else {
128       struct frame reply;
129       Return_e ret = f_receive_frame(conn->fd, &reply, 30);
130       if (ret == ERROR_CODE) {
131         fprintf(stderr, "ret = %d\n", ret);
132         printf("line: %d\n", __LINE__);
133         return 1;
134       }
135       if (reply.control == C_SET) {
136         f_send_frame(conn->fd, UA);
137       }
138     }
139     return 0;
140   }
141
142   int test_3(struct connection* conn)
143   {
144     int test_timeout_time = 3;
145
146     if (TRANSMITTER) {
147       f_print_frame(SET);
148       f_dump_frame_buffer("FRAME");
149       struct frame reply;
150       if (0
151           > f_send_acknowledged_frame(conn->fd, 3,
                test_timeout_time, SET,
152               &reply)) {
153         printf("line: %d\n", __LINE__);
154         return 1;
155       }
156       if (reply.control != C_UA) {
157         printf("line: %d\n", __LINE__);
```

```
158         return 1;
159       }
160     } else {
161       printf("Sleeping for %d seconds...", test_timeout_time +
              1);
162       sleep(test_timeout_time + 1); // force timeout
163
164       struct frame reply;
165       Return_e ret;
166
167       ret = f_receive_frame(conn->fd, &reply, 3);
168       if (ret == ERROR_CODE) {
169         fprintf(stderr, "ret = %d\n", ret);
170         printf("line: %d\n", __LINE__);
171         return 1;
172       }
173       ret = f_receive_frame(conn->fd, &reply, 3);
174       if (ret == ERROR_CODE) {
175         fprintf(stderr, "ret = %d\n", ret);
176         printf("line: %d\n", __LINE__);
177         return 1;
178       }
179       if (reply.control == C_SET) {
180         f_send_frame(conn->fd, UA);
181       }
182     }
183     return 0;
184   }
185
186   int test_4(struct connection *conn)
187   {
188     if (TRANSMITTER) {
189       if (transmitter_connect(conn) < 0) {
190         printf("line: %d\n", __LINE__);
191         return 1;
192       }
193       printf("Connection established.\n");
194       if (disconnect(conn) < 0) {
195         printf("line: %d\n", __LINE__);
196         return 1;
197       }
198     } else {
199       if (receiver_listen(conn) < 0) {
200         printf("line: %d\n", __LINE__);
201         return 1;
202       }
203       if (wait_for_disconnect(conn, 0) < 0) {
204         printf("line: %d\n", __LINE__);
205         return 1;
206       }
207     }
208     return 0;
209   }
210
```

```
211 int test_single_message(struct connection *conn, byte* data)
212 {
213   if (TRANSMITTER) {
214     if (transmitter_connect(conn) < 0) {
215       printf("line: %d\n", __LINE__);
216       return 1;
217     }
218
219     byte* s = data;
220     if (transmitter_write(conn, s, strlen((char*) s) + 1) <
          0) {
221       printf("line: %d\n", __LINE__);
222       return 1;
223     }
224     printf("--- Transmitted: %s\n", (char*) s);
225
226     if (disconnect(conn) < 0) {
227       printf("line: %d\n", __LINE__);
228       return 1;
229     }
230   } else {
231     if (receiver_listen(conn) < 0) {
232       printf("line: %d\n", __LINE__);
233       return 1;
234     }
235
236     byte dest[8000];
237
238     if (receiver_read(conn, dest, 8000, 0) < 0) {
239       printf("line: %d\n", __LINE__);
240       return 1;
241     }
242     printf("--- Received: %s\n", (char*) dest);
243     if (strcmp((char*) dest, (char*) data) != 0) {
244       printf("%s\n", (char*) dest);
245       printf("%s\n", (char*) data);
246       printf("line: %d\n", __LINE__);
247       return 1;
248     }
249   }
250   return 0;
251 }
252
253 int test_5(struct connection* conn)
254 {
255   char *data = "isto e um teste";
256   return test_single_message(conn, (byte*) data);
257 }
258
259 int test_6(struct connection* conn)
260 {
261   char data[] = "Flag: x, Escape: x";
262   data[6] = FLAG;
263   data[17] = BS_ESC;
```

```
264   return test_single_message(conn, (byte*) data);
265 }
266
267 int send_message(struct connection* conn, byte* s)
268 {
269   int len = strlen((char*) s);
270   if (len == 0) {
271     len = 1;
272   }
273   if (transmitter_write(conn, s, len) < 0) {
274     printf("line: %d\n", __LINE__);
275     return 1;
276   }
277   printf("--- Transmitted: %s\n", (char*) s);
278   return 0;
279 }
280
281 int test_7(struct connection* conn)
282 {
283   char data1[] = "isto ";
284   char data2[] = "e ";
285   char data3[] = "um ";
286   char data4[] = "teste ";
287   char data5[] = "com ";
288   char data6[] = "varias ";
289   char data7[] = "tramas.";
290   char data8[] = "";
291   char final_string[] = "isto e um teste com varias tramas."
        ;
292
293   printf("data1 = %s\n", data1);
294
295   if (TRANSMITTER) {
296     if (transmitter_connect(conn) < 0) {
297       printf("line: %d\n", __LINE__);
298       return 1;
299     }
300     send_message(conn, (byte*) data1);
301     send_message(conn, (byte*) data2);
302     send_message(conn, (byte*) data3);
303     send_message(conn, (byte*) data4);
304     send_message(conn, (byte*) data5);
305     send_message(conn, (byte*) data6);
306     send_message(conn, (byte*) data7);
307     send_message(conn, (byte*) data8);
308
309     if (disconnect(conn) < 0) {
310       printf("line: %d\n", __LINE__);
311       return 1;
312     }
313   } else {
314     if (receiver_listen(conn) < 0) {
315       printf("line: %d\n", __LINE__);
316       return 1;
```

```
317       }
318
319       byte dest[8000];
320
321       if (receiver_read(conn, dest, 8000, 0) < 0) {
322         printf("line: %d\n", __LINE__);
323         return 1;
324       }
325       printf("--- Received: %s\n", (char*) dest);
326       if (strcmp((char*) dest, final_string) != 0) {
327         printf("%s\n", (char*) dest);
328         printf("%s\n", final_string);
329         printf("line: %d\n", __LINE__);
330         return 1;
331       }
332     }
333     return 0;
334   }
335
336   int main(int argc, char *argv[])
337   {
338     int i;
339     if ((i = parse_args(argc, argv)) < 0) {
340       help(argv);
341       printf("line: %d\n", __LINE__);
342       return 1;
343     }
344
345     struct connection conn = CONNECTION;
346
347     strcpy(conn.port, argv[i]);
348
349     if ((conn.fd = serial_port_open(conn.port, conn.
          micro_timeout_ds)) < 0) {
350       printf("line: %d\n", __LINE__);
351       return 1;
352     }
353     assert(test_1(&conn) == 0);
354     printf(" ------------ Passed test 1\n");
355     assert(test_2(&conn) == 0);
356     printf(" ------------ Passed test 2\n");
357     assert(test_3(&conn) == 0);
358     printf(" ------------ Passed test 3\n");
359     if (serial_port_close(conn.fd, 3) < 0) {
360       printf("line: %d\n", __LINE__);
361       return 1;
362     }
363
364     assert(test_4(&conn) == 0);
365     printf(" ------------ Passed test 4\n");
366     assert(test_5(&conn) == 0);
367     printf(" ------------ Passed test 5\n");
368     assert(test_6(&conn) == 0);
369     printf(" ------------ Passed test 6\n");
```

```
370   assert(test_7(&conn) == 0);
371   printf(" ------------ Passed test 7\n");
372   printf("Finished\n");
373   return 0;
374 }
```

serial_port_test.c

```
1  #include <fcntl.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include <stdlib.h>
5  #include <unistd.h>
6  #include <assert.h>
7
8  #include "serial_port.h"
9  #include "data_link.h"
10 #include "byte.h"
11
12 #define FALSE 0
13 #define TRUE 1
14
15 #define BUFSIZE 4096
16
17 int is_transmitter = FALSE;
18
19 // Print how the arguments must be
20 void help(char **argv)
21 {
22   printf("Usage: %s [OPTION] <serial port>\n", argv[0]);
23   printf("\n Program options:\n");
24   printf("  -t         transmit data over the serial port\n"
       );
25 }
26
27 // Verifies serial port argument
28 int parse_serial_port_arg(int index, char **argv)
29 {
30     char *dev = argv[index];
31   if ( (strcmp("/dev/ttyS0", dev)!= 0) &&
32     (strcmp("/dev/ttyS1", dev)!=0) &&
33     (strcmp("/dev/ttyS4", dev)!=0) ) {
34     return -3;
35     }
36   return index;
37 }
38
39 // Verifies arguments
40 int parse_args(int argc, char **argv)
41 {
42   if (argc < 2)
43     return -1;
44
45   if (argc == 2)
```

```
46      return parse_serial_port_arg(1, argv);
47
48    if (argc == 3) {
49      if ( (strcmp("-t", argv[1]) != 0) )
50        return -2;
51      else is_transmitter = TRUE;
52
53      return parse_serial_port_arg(2, argv);
54    }
55    else return -1;
56  }
57
58  int send_receive_test(char* port,byte* test_message)
59  {
60      int timeout = 0;
61      int fd = serial_port_open(port,timeout);
62      if (fd < 0) {
63          fprintf(stderr,"serial_port_test: serial_port_open
                returned %d\n",fd);
64          printf("line: %d\n",__LINE__);
65          return 1;
66      }
67
68      byte s[BUFSIZE];
69      if (is_transmitter) {
70          int len = strlen((char*)test_message);
71          if (serial_port_write(fd,(byte*)test_message,len+1)
                < 0) {
72              printf("line: %d\n",__LINE__);
73              return 1;
74          }
75          printf("len = %d\n",len);
76          //printf("Message sent: %s\n",s);
77
78          byte s[BUFSIZE];
79          for (int i=0; i < BUFSIZE; i++) {
80              s[i] = 1;
81          }
82          int r = serial_port_read(fd,s,'\0',BUFSIZE);
83          if (r <= 0) {
84              printf("r = %d\n",r);
85              printf("line: %d\n",__LINE__);
86              return 1;
87          }
88          //printf("Message received: %s\n",s);
89
90    if (strcmp((char*)test_message,(char*)s) != 0) {
91          printf("r = %d\n",r);
92      printf("Test failed\n");
93          /*
94          for (int i = 0; test_message[i] != '\0'; ++i) {
95              if (test_message[i] != s[i]) {
96                  printf("test_message %x\n",test_message[i]);
97                  printf("s %x\n",s[i]);
```

```
98              }
99          }
100         */
101     return -1;
102   }
103
104     } else {
105         if (serial_port_read(fd,s,'\0',BUFSIZE) < 0) {
106             printf("line: %d\n",__LINE__);
107             return 1;
108         }
109         //printf("Message received: %s\n",s);
110
111         int len = strlen((char*)s);
112         if (serial_port_write(fd,s,len+1) < 0) {
113             printf("line: %d\n",__LINE__);
114             return 1;
115         }
116         //printf("Message sent: %s\n",(char*)s);
117     }
118
119     if (serial_port_close(fd,3) < 0) {
120         fprintf(stderr,"serial_port_test: serial_port_close
                returned \
121                 negative\n");
122         printf("line: %d\n",__LINE__);
123         return 1;
124     }
125     return 0;
126 }
127
128
129 int test1(int argc,char **argv)
130 {
131      // Verifies arguments
132      int i = -1;
133      if ( (i = parse_args(argc, argv)) < 0 ) {
134       help(argv);
135              printf("line: %d\n",__LINE__);
136       return 1;
137      }
138
139     if (send_receive_test(argv[i],(byte*)"Um pequeno passo
            para o homem...") == 0) {
140         printf("Test 1 passed\n");
141     }
142     return 0;
143 }
144
145 int get_frame(byte *dest,byte *src,byte flag)
146 {
147     for (int i = 0; i < BUFSIZE; i++) {
148         if (src[i] == flag) {
149             dest[i] = '\0';
```

```
150             return i;
151         }
152         dest[i] = src[i];
153     }
154     return -1;
155 }
156
157 int test2(int argc,char **argv)
158 {
159     // Verifies arguments
160     int i = -1;
161     if ((i = parse_args(argc, argv)) < 0 ) {
162         help(argv);
163         printf("line: %d\n",__LINE__);
164        return 1;
165     }
166
167     int timeout = 0;
168     int fd = serial_port_open(argv[i],timeout);
169     if (fd < 0) {
170         fprintf(stderr,"serial_port_test: serial_port_open
                returned %d\n",fd);
171         printf("line: %d\n",__LINE__);
172         return 1;
173     }
174
175
176     if (is_transmitter) {
177         byte *test_string = (byte*) "
                F0001FF0002F0003F0004F  ";
178
179         int len = strlen((char*)test_string);
180         if (serial_port_write(fd,test_string,len+1) < 0) {
181             printf("line: %d\n",__LINE__);
182             return 1;
183         }
184     } else {
185         // Read until first flag
186         byte tmp[BUFSIZE];
187         for (int i = 0; i < BUFSIZE; i++) { // init
188             tmp[i] = 'x';
189         }
190         if (serial_port_read(fd,tmp,'F',BUFSIZE) < 0) {
191             printf("line: %d\n",__LINE__);
192             return 1;
193         }
194
195         byte dest[BUFSIZE];
196         for (int i = 0; i < BUFSIZE; i++) { // init
197             dest[i] = 'x';
198         }
199
200         byte *frames[] =
```

```
201         { (byte*)"0001", (byte*)"0002", (byte*)"0003", (byte
                *)"0004" };
202
203         // Read a few frames
204         for (int i = 0; i < 4; i++) {
205             // Read a frame
206             if (serial_port_read(fd,tmp,'F',BUFSIZE) < 0) {
207                 printf("line: %d\n",__LINE__);
208                 return 1;
209             }
210
211             byte plb = serial_port_previous_last_byte();
212             printf("previous last byte: %c\n",plb);
213             if (plb != 'F') {
214                 printf("line: %d\n",__LINE__);
215                 return 1;
216             }
217
218             int size = get_frame(dest,tmp,'F');
219             if (size < 0) {
220                 printf("line: %d\n",__LINE__);
221                 return 1;
222             } else if (size == 0) {
223                 // Space in between 'F's
224                 --i;
225             } else {
226                 printf("frame %d: %s\n",i,(char*)dest);
227                 if (strcmp((char*)dest,(char*)frames[i]) !=
                        0) {
228                     printf("line: %d\n",__LINE__);
229                     return 1;
230                 }
231             }
232         }
233     }
234
235     if (serial_port_close(fd,3) < 0) {
236         fprintf(stderr,"serial_port_test: serial_port_close
                returned \
237                 negative\n");
238         return 1;
239     }
240
241     printf("Test 2 passed\n");
242     return 0;
243 }
244
245 int test3(int argc,char **argv)
246 {
247     // Verifies arguments
248     int i = -1;
249     if ( (i = parse_args(argc, argv)) < 0 ) {
250         help(argv);
251         printf("line: %d\n",__LINE__);
```

```
252             return 1;
253         }
254
255         byte message[256];
256         for (int j = 0; j < 256; ++j) {
257           message[j] = j+1;
258         }
259         message[255] = '\0';
260         //message[0] = 100;
261         //message[1] = 101;
262         //message[2] = 0;
263
264         if (send_receive_test(argv[i],message) == 0) {
265             printf("Test 3 passed\n");
266         }
267         return 0;
268     }
269
270     int main(int argc, char **argv)
271     {
272         //printf("Test 1...\n");
273         //assert(test1(argc,argv) == 0);
274
275         //printf("Test 2...\n");
276         //assert(test2(argc,argv) == 0);
277
278         printf("Test 3...\n");
279         assert(test3(argc,argv) == 0);
280       return 0;
281     }
```