Redes de Computadores

Ligação de Dados

Nuno Miguel Fernandes Marques - 201708997 - MIEIC

November 14, 2020

Sumário

Este relatório e feito no âmbito do primeiro trabalho laboratorial de Redes de computadores. O trabalho consiste na transmissão de ficheiros usando a porta de série. As principais conclusões obtidas neste projeto foram:

- As relações de tamanho da trama e erro em tramas com a eficiência da ligação.
- A eficiência da transmissão de dados pela porta de séria, até 79% neste projeto.
- A necessidade de camadas independentes na transmissão de dados em redes de computadores.

Introdução

Este projeto tinha como objectivo final desenvolver e testar uma aplicação com objectivo de realizar a **transmissão de ficheiros pela porta de série** fazendo o uso de duas camadas independentes, a ligação de dados e a aplicação. Este relatório está divido nas seguintes secções:

- **Arquitetura** - Blocos funcionais e interfaces
- **Estrutura do código** - API, estruturas de dados e funções
- **Casos de uso principais** - Casos de uso do projeto e sequências de chamada de funções
- **Protocolo de ligação lógica** - Descrição dos aspetos funcionais e estratégias usadas na camada da ligação
- **Protocolo de aplicação**- Descrição dos aspetos funcionais e estratégias usadas na camada da aplicação
- **Validação** - Validações efectuadas
- **Eficiência do protocolo de ligação de dados** - Medição e análise da eficiência do protocolo usado
- **Conclusões** - Resumo das conclusões sobre o projeto e sobre o tema apresentado.

Arquitetura

Existem dois **blocos funcionais** independentes no trabalho, o bloco da **ligação de dados** e o bloco da **aplicação**. O bloco de ligação de dados tem como objectivo abrir e fechar a ligação, assegurar a transmissão correta das tramas e recuperação quando ocorrem erros de transmissão. Este bloco tem um nível baixo fazendo a ligação diretamente com a porta de série.

O Bloco da aplicação tem como objectivo ler e escrever ficheiros, gerir o tamanho das tramas de dados e assegurar que os dados recebidos são válidos. Este bloco tem um nível mais alto fazendo uso da API definida no bloco da ligação de dados.

Ambos os blocos fazem uso de um **array dinâmico** na sua gestão de memória.

A **interface** permite o uso do mesmo executável para recepção e transmissão, especificando a opção na linha de comandos. Alem disso e necessário especificar o nome do ficheiro a transmitir e a porta de série a usar. Baudrate, tamanho máximo das tramas, número máximo de tentativas de retransmissão e tempo limite sem comunicação são definições opcionais. O progresso da transmissão e mostrada usando uma barra de progresso. No fim são apresentadas as estatísticas da transmissão.

```
al: waiting for connection
al: starting file transmission
Progress ##########[100.00]
al: file transmission is over
Statistics:
 baudrate 38400 bits/s  average bitrate 30450 bits/s    Efficiency 79.30%
 file size 10968 bytes  max fragment size 65535 bytes   packets sent 0
 transmission time 2.88 seconds
 total Frames 4         lost frames 0   frame loss 0.00
```

Exemplo do output de uma execução

Estrutura do código

Camada de ligação

A *API* da camada de ligação disponibiliza as seguintes funções:

```
void llabort(int fd);
void ll_setup(int timeout, int max_retries, int baudrate);
int llopen(int port, link_type type);
int llclose(int fd);
int llwrite(int fd, char* buffer, int length);
int llread(int fd, char** buffer);
ll_statistics ll_get_stats();
```

E internamente usa as seguintes estruturas de dados:

```
typedef struct {
  char port[MAX_PORT_LENGTH];
  int baud_rate;
  unsigned int sequence_number;
  unsigned int timeout;
  unsigned int num_transmissions;
} link_layer;
```

```
typedef enum {
  FD_FIELD = 0x00,    // Start Flag
  AF_FIELD = 0x01,    // Address
  C_FIELD = 0x02,     // Control
  BCC1_FIELD = 0x03,  // BCC1
  DATA_FIELD = 0x04   // Data field start
} ll_control_frame_field;
```

Camada da Aplicação

A camada de ligação disponibiliza as seguintes funções:

```
void al_setup(int timeout, int baudrate, int max_retries, int frag_size);
int al_sendFile(const char *filename, int port);
int al_receiveFile(const char *filename, int port);
al_statistics al_get_stats();
void al_print_stats();
```

E internamente usa as seguintes estruturas de dados:

```
typedef struct {
  control_type type;
  int8_t sequenceNr;
  uint16_t size;
  char *data;
} data_packet;
```

```
typedef struct {
  control_type type;
  char *name;
  int8_t nameLength;
  uint32_t size;
  uint8_t sizeLength;
} control_packet;
```

Casos de uso principais

A **camada de ligação** terá como casos de uso principais permitir uma aplicação **ligar** a porta de série**(llopen)**, **enviar ou receber** segmentos de dados**(llread e llwrite)** e **desligar a ligação(llclose)**. Alem disso disponibiliza estatísticas sobre a ligação, neste caso tramas totais enviadas e tramas perdidas. Esta camada pode ser configurada com uso da função ***ll_setup***, sendo possível alterar o *baudrate*, o número máximo de retransmissões e o tempo limite para retransmissão.

A camada da aplicação tem dois casos de uso principais: **enviar ou receber** um ficheiro usando as funções **al_sendFile** e **al_receiveFile** respectivamente. Esta camada pode ser configurada com o uso da função **al_setup** que permite configurar o tamanho máximo de cada segmento de dados individual. Esta camada disponibiliza também estatísticas sobre bits por segundo médio, quantidade de segmentos de dados enviados, duração da transmissão e eficiência em relação ao valor máximo teórico de transmissão.

No caso de receção a função **al_receiveFile** faz uso das funções: **llopen**, **llread** e **llclose** da camada de ligação.
No caso de emissão a função **al_sendFile** faz uso das funções **llopen**, **llwrite** e **llclose** da camada de ligação

Protocolo de ligação lógica

O protocolo de ligação de dados começa pela função **llopen**, esta função começa por abrir a ligação com a porta de série usando as configuração fornecidas. Após estabelecer a ligação, o receptor espera que o emissor envie uma trama de controlo SET, ao enviar a trama de controlo SET, o emissor fica a espera de uma trama de controlo UA como resposta.

Após essa comunicação é possível começar a transmissão de tramas de dados usando **(llread e llwrite)**. Em **llwrite** o cabeçalho da trama de dados é criado, depois o segmento de dados é introduzido na trama byte a byte. Simultâneamente e calculado o byte de segurança **BCC2** e é feito o **byte stuffing**, finalmente é verificado se o **BCC2** precisa **byte stuffing** e é introduzido na trama. Com a trama completa é feito o envio pela porta de séria, uma trama de controlo RR é esperada em resposta. Caso a resposta demore mais que o tempo limite, a trama RR seja duplicada ou seja uma

resposta do tipo REJ a retransmissão da trama é feita. No caso de ultrapassar o limite de retransmissões a operação é abortada.

```
// Add buffer to frame and calculate bcc2
uchar_t bcc2 = 0x00;
for (int i = 0; i < length; ++i) {
  // BCC2
  bcc2 ^= (uchar_t)buffer[i];
  // Byte stuffing
  if (buffer[i] == (uchar_t)LL_FLAG || buffer[i] == (uchar_t)LL_ESC) {
    char_buffer_push(frame, (uchar_t)LL_ESC);
    char_buffer_push(frame, buffer[i] ^ (uchar_t)LL_ESC_MOD);
  } else
    char_buffer_push(frame, buffer[i]);
}

// Bytestuffin on BBC2 when needed
if (bcc2 == (uchar_t)LL_ESC || bcc2 == (uchar_t)LL_FLAG) {
  char_buffer_push(frame, (uchar_t)LL_ESC);
  char_buffer_push(frame, (uchar_t)(bcc2 ^ (uchar_t)LL_ESC_MOD));
} else
  char_buffer_push(frame, bcc2);
```

Calculo de BCC2 e byte stuffing

Em **llread** é esperada uma trama de dados. Ao receber a trama o cabeçalho é validado, e os dados são lidos byte a byte, fazendo o **byte destuffing** quando necessário e calculando o **BCC2** esperado ao mesmo tempo. Depois deste processo o **BCC2** é lido e comparado com o **BCC2** esperado. Se este processo for feito sem problemas uma trama de controlo RR é enviada. Caso seja encontrado um problema em qualquer dos passos uma trama de controlo REJ é enviada.

Finalmente em **llclose** o emissor envia uma trama de controlo DISC, á qual o receptor responde também com DISC. Neste ponto o emissor envia a trama de controlo UA e encerra a ligação e repõe a configuração da porta de série. O receptor faz o mesmo após receber a trama de controlo UA.

Protocolo de aplicação

O protocolo de aplicação tem um nível mais alto e faz uso da API do protocolo de ligação de dados para efectuar transferências de ficheiros. Esta camada pode ser representa por apenas duas funçoes: **al_receiveFile e al_sendFile**. Ambas as funções inicialização e fecham a ligação no fim da operação.

A função **al_sendFile** lê o ficheiro e primeiro envia um pacote de controlo para sinalizar o inicio da transferência. Depois para cada segmento de dados é criado um pacote de dados que inclui o nome do ficheiro, número de sequência do pacote, tamanho do segmento e o próprio segmento, usando a função llwrite da camada de ligação. Quando todos segmentos forem enviados, outro pacote de controlo é enviado para sinalizar o fim da transferência.

A função **al_receiveFile** usa a função llread para ler estes pacotes. Á medida que recebe pacotes a função guarda os segmentos num ficheiro até receber o pacote de controlo a sinalizar o fim da transmissão. Esta função valida os números de sequência dos pacotes e, no fim, se recebeu a quantidade correta de bytes.

Esta camada também calcula e disponibiliza **estáticas** da transmissão.

Validação

Os seguintes testes foram aplicados neste trabalho:

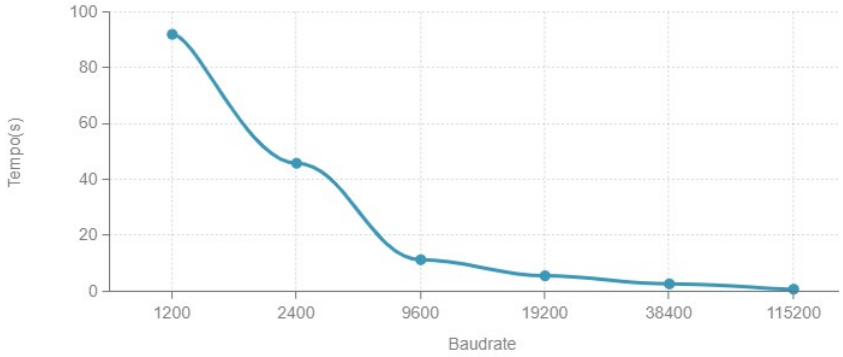
- Envio de ficheiros de tamanhos diferentes
- Interrupção de ligação em momentos aleatórios durante a transmissão
- Introdução de bits aleatórios durante a transmissão
- Simulação de erros no BCC2 das tramas de dados
- Variação do baudrate, número máximo de retransmissões, tempo até retransmissão e tamanho da trama

Pelo fim do projeto os testes foram realizados com sucesso.

Eficiência do protocolo de ligação de dados

Neste testes a efeciencia foi medida usando a formula *S=(R/C)*.

Variável: Capacidade de ligação (C)



Testes realizados com ficheiro de tamanho constante 10.7KB, com trama constante de 10KB. A **eficiência** manteve estável em todos testes, sendo sempre aproximadamente **79%**.

Variável: Tamanho da trama de dados

Tamanho(bytes)	Tempo(s)	R(bits/s)	S(R/C)(%)
10	8.60	10199	23.56
40	4.31	20359	53.02
100	3.44	25486	66.37
500	2.99	29348	76.43
1000	2.93	29917	77.91
4000	2.89	30340	79.01
10000	2.88	30416	79.21

Nestes testes foi usado um *baudrate* constante de 38400 e uma imagem de tamanho constante 10.7KB. Nestes testes observamos que existe um ganho grande de eficiência a aumentar o tamanho da trama, para tamanhos pequenos. Mas a partir dos **500 *bytes*** os ganhos começam a ficar insignificantes.

Variável:*Frame Error Ratio* (FER)

Probabilidade Erro(%)	S(R/C)(%)	Perda de tramas registada(%)
0	73.66	0
1	73.42	2
5	70.40	4
10	67.38	8
20	60.93	16
50	34.55	54

Nestes testes foi usado um *baudrate* constante de 38400, uma imagem de tamanho constante 10.7KB e trama de dados de tamanho fixo a 256 *bytes*.

Foram introduzido erros artificiais no calculo do BCC2, que produz uma resposta REJ e uma retransmissão imediata da trama. Podemos concluir que existe uma **relação proporcional** entre a **eficiência** e a **perda de tramas**.

Conclusões

Este trabalho consistiu na criação de duas **camadas independentes**, uma camada de ligação e uma camada da aplicação, com o objectivo de efectuar a transmissão de ficheiros via porta de série. Este projeto deu a conhecer diretamente as diferentes camadas de comunicação, protocolos de comunicação, validação e recuperação de erros e o calculo de eficiência presentes nas redes de computadores.

O projeto foi completado com sucesso apresentando **79% de eficiência**, em situações ideais, e validação e recuperação correta de erros em situações menos ideais.

Anexo I

main.c

```
1
2
3  #include "link_layer.h"
4  #include "app_layer.h"
5  #include "char_buffer.h"
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <unistd.h>
9  #include <string.h>
10 #include <time.h>
11
12 #define DEFAULT_MAX_TRANSMISSION_ATTEMPS 3
13 #define DEFAULT_TIMEOUT_DURATION 3
14 #define DEFAULT_BAUDRATE 38400
15 #define MAX_FRAGMENT_SIZE 0xFFFF
16
17 void print_usage(const char* arg) {
18   printf("Usage:\n");
19   printf("%s <file_name> <T|R> <port_number> [options]\n", arg)
       ;
20   printf("T - Transmitter, R - Receiver\n");
21   printf("Options:\n");
22   printf("  -timeout=<seconds> \t\tSeconds until a frame is
       timed out\n");
23   printf("  -baudrate=<rate> \t\tSerial port rate\n");
24   printf(
25       "  -max_retries=<retries> \tTimes a frame transmission
       can be "
26       "retried\n");
27   printf("  -frag_size=<size> \t\tMax size for data fragments\n
       ");
28   printf("\nExample: '%s pinguim.gif T 10'\n", arg);
29 }
30
31 int main(int argc, char** argv) {
32   if (argc < 4) {
33     print_usage(argv[0]);
34     return -1;
35   }
36   srand(time(0));
37
38   char* file_name = argv[1];  // File name
39   // Read link type
40   link_type type = RECEIVER;
41   if (argv[2][0] == 'T') type = TRANSMITTER;
```

```
42
43    int port = atoi(argv[3]);  // Read Port
44
45    /* Options */
46    int timeout = DEFAULT_TIMEOUT_DURATION;
47    int retries = DEFAULT_MAX_TRANSMISSION_ATTEMPS;
48    int baudrate = DEFAULT_BAUDRATE;
49    int frag_size = MAX_FRAGMENT_SIZE;
50
51    for (int i = 4; i < argc; ++i) {
52      if (!strncmp(argv[i], "-timeout=", 9) && strlen(argv[i]) >
       9) {
53        timeout = atoi(&argv[i][9]);
54        continue;
55      }
56
57      if (!strncmp(argv[i], "-max_retries=", 13) && strlen(argv[i
       ]) > 13) {
58        retries = atoi(&argv[i][13]);
59        continue;
60      }
61
62      if (!strncmp(argv[i], "-baudrate=", 10) && strlen(argv[i])
       > 10) {
63        baudrate = atoi(&argv[i][10]);
64        continue;
65      }
66
67      if (!strncmp(argv[i], "-frag_size=", 11) && strlen(argv[i])
       > 11) {
68        frag_size = atoi(&argv[i][11]);
69        continue;
70      }
71    }
72
73    al_setup(timeout, baudrate, retries, frag_size);
74
75    int res;
76    if (type == RECEIVER) {
77      res = al_receiveFile(file_name, port);
78    } else {
79      res = al_sendFile(file_name, port);
80    }
81
82    if (res >= 0) al_print_stats();
83
84    return 0;
85  }
```

app_layer.h

```
#ifndef APP_LAYER_H
#define APP_LAYER_H

#include <stdint.h>
#include <stdio.h>

typedef struct {
  unsigned int baudrate;
  unsigned int timeout;
  unsigned int retries;
  unsigned int avg_bits_per_second;
  unsigned int data_packet_count;
  unsigned int file_size;
  unsigned int frames_total;
  unsigned int frames_lost;
  double transmission_duration_secs;
} al_statistics;

void al_setup(int timeout, int baudrate, int max_retries, int
    frag_size);
int al_sendFile(const char *filename, int port);
int al_receiveFile(const char *filename, int port);
al_statistics al_get_stats();
void al_print_stats();

#endif
```

app_layer.c

```c
#include "app_layer.h"
#include "link_layer.h"
#include "char_buffer.h"

#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <stdbool.h>
#include <sys/time.h>
#include <time.h>

//#define AL_PRINT_CPACKETS

#define MAX_FRAGMENT_SIZE 0xFFFF
#define MAX_BAUDRATE 460800
#define DATA_HEADER_SIZE 4
#define MAX_FILE_NAME 256

#define CP_CFIELD 0x00
#define SEQ_FIELD 0x01
#define L2_FIELD 0x02
#define L1_FIELD 0x03

#define TLV_SIZE_T 0x00
#define TLV_NAME_T 0x01
#define CP_MIN_SIZE 7

#define AL_LOG_INFORMATION

typedef unsigned char uchar_t;
typedef enum {
  CONTROL_START = 0x02,
  CONTROL_END = 0x03,
  CONTROL_DATA = 0x01
} control_type;

typedef struct {
  control_type type;
  char *name;
  int8_t nameLength;
  uint32_t size;
  uint8_t sizeLength;
} control_packet;

typedef struct {
  control_type type;
  int8_t sequenceNr;
```

```
48   uint16_t size;
49   char *data;
50 } data_packet;
51
52 control_packet fileCP;  // Control Packet with file information
53 static al_statistics al_stats;
54 static int al_frag_size = MAX_FRAGMENT_SIZE;
55
56 void print_control_packet(control_packet *packet);
57 int read_control_packet(int fd, control_packet *packet);
58 int parse_control_packet(char *packetBuffer, int size,
       control_packet *cp);
59 int send_control_packet(int fd, control_type type);
60 void build_control_packet(control_type type, char_buffer *
       packet);
61 int get_file_info(const char *filename, FILE *fptr);
62 int read_data_packet(int fd, data_packet *packet, char *buffer)
       ;
63 int send_data_packet(int fd, data_packet *packet);
64 void print_progress(int done, int total);
65
66 void al_log_msg(const char *msg) {
67 #ifdef AL_LOG_INFORMATION
68   fprintf(stderr, "al: %s\n", msg);
69 #endif
70 }
71
72 float clock_seconds_since(struct timeval *start_timer) {
73   struct timeval end_timer;
74   gettimeofday(&end_timer, NULL);
75   float elapsed = (end_timer.tv_sec - start_timer->tv_sec);
76   elapsed += (end_timer.tv_usec - start_timer->tv_usec) /
       1000000.0f;
77   return elapsed;
78 }
79
80 void update_statistics(struct timeval *start_timer) {
81   al_stats.file_size = fileCP.size;
82   al_stats.transmission_duration_secs = clock_seconds_since(
       start_timer);
83   ll_statistics ll_stats = ll_get_stats();
84   al_stats.frames_total = ll_stats.frames_total;
85   al_stats.frames_lost = ll_stats.frames_lost;
86   al_stats.avg_bits_per_second =
87       (float)(al_stats.file_size * 8) / al_stats.
       transmission_duration_secs;
88 }
89
90 al_statistics al_get_stats() { return al_stats; }
```

```
91
92 void al_print_stats() {
93   printf("Statistics:\n");
94   float eff =
95       (float)al_stats.avg_bits_per_second / (float)al_stats.
     baudrate * 100.0f;
96   printf(
97       " baudrate %d bits/s \taverage bitrate %d bits/s \
     tEfficiency %.2f%% \n",
98       al_stats.baudrate, al_stats.avg_bits_per_second, eff);
99   printf(" file size %d bytes \tmax fragment size %d bytes \
     tpackets sent %d\n",
100          al_stats.file_size, al_frag_size, al_stats.
     data_packet_count);
101  printf(" transmission time %.2f seconds\n",
102         al_stats.transmission_duration_secs);
103  float floss = (float)al_stats.frames_lost / (float)al_stats.
     frames_total;
104  printf(" total Frames %d \tlost frames %d \tframe loss %.2f\n
     ",
105         al_stats.frames_total, al_stats.frames_lost, floss);
106 }
107
108 void al_setup(int timeout, int baudrate, int max_retries, int
     frag_size) {
109  al_stats.timeout = timeout;
110  al_stats.retries = max_retries;
111  al_stats.data_packet_count = 0;
112  if (baudrate > MAX_BAUDRATE) baudrate = MAX_BAUDRATE;
113  al_stats.baudrate = baudrate;
114  al_frag_size = frag_size;
115  if (al_frag_size > MAX_FRAGMENT_SIZE) al_frag_size =
     MAX_FRAGMENT_SIZE;
116  ll_setup(timeout, max_retries, baudrate);
117 }
118
119 int al_sendFile(const char *filename, int port) {
120   int nameLength = strlen(filename);
121   if (nameLength > MAX_FILE_NAME) {
122     al_log_msg("Filename length exceeds limits(256 characters)"
     );
123     return -1;
124   }
125   fileCP.nameLength = nameLength;
126
127   // Open File Stream
128   FILE *fptr = fopen(filename, "r");
129   if (fptr == NULL) {
130     al_log_msg("Could not open selected file");
```

```
    return -1;
  }

  // Establish LL Connection
  int fd = llopen(port, TRANSMITTER);
  if (fd == -1) {
    al_log_msg("Failed to establish connection");
    return -1;
  }

  // Get File Information
  get_file_info(filename, fptr);

  // Send start control packet
  if (send_control_packet(fd, CONTROL_START) == -1) return -1;

  struct timeval start_timer;
  gettimeofday(&start_timer, NULL);
  printf("al: starting file transmission\n");

  // Send data packets until the file is read
  data_packet packet;
  packet.data = (char *)malloc(al_frag_size + DATA_HEADER_SIZE)
    ;
  packet.sequenceNr = 0;
  packet.size = 1;
  unsigned int bytesTransferred = 0;
  while (true) {
    packet.size =
        fread(&packet.data[L1_FIELD + 1], sizeof(uchar_t),
    al_frag_size, fptr);
    if (packet.size <= 0) {
      break;
    }
    ++al_stats.data_packet_count;
    ++packet.sequenceNr;
    packet.sequenceNr %= 256;
    if (send_data_packet(fd, &packet) == -1) {
      al_log_msg("file transmission failed, aborting...");
      llabort(fd);
      return -1;
    }
    // Progress
    bytesTransferred += packet.size;
    print_progress(bytesTransferred, fileCP.size);
  }
  printf("\n");
  free(packet.data);

```

```
178   // Send end control packet
179   if (send_control_packet(fd, CONTROL_END) == -1) return -1;
180   printf("al: file transmission is over\n");
181   update_statistics(&start_timer);
182   // Close connection and cleanup
183   llclose(fd);
184   free(fileCP.name);
185   fclose(fptr);
186
187   return 0;
188 }
189
190 int al_receiveFile(const char *filename, int port) {
191   printf("al: waiting for connection\n");
192
193   static int8_t seq_number = 1;
194
195   // Establish LL Connection
196   int fd = llopen(port, RECEIVER);
197   if (fd == -1) {
198     al_log_msg("Failed to establish connection");
199     return -1;
200   }
201
202   FILE *fptr = fopen(filename, "w");
203   if (fptr == NULL) {
204     al_log_msg("Could not write selected file");
205     return -1;
206   }
207
208   fileCP.type = CONTROL_DATA;
209   while (fileCP.type != CONTROL_START) {
210     read_control_packet(fd, &fileCP);
211   }
212
213   struct timeval start_timer;
214   gettimeofday(&start_timer, NULL);
215   printf("al: starting file transmission\n");
216
217   unsigned int bytesTransferred = 0;
218   data_packet data_packet;
219   while (bytesTransferred < fileCP.size) {
220     char *buffer = NULL;
221     if (read_data_packet(fd, &data_packet, buffer) == -1)
      return -1;
222
223     if (data_packet.sequenceNr != seq_number) {
224       al_log_msg("ignoring duplicate data packet");
225       free(buffer);
```

```
      continue;
    }
    ++seq_number;
    seq_number %= 256;

    fwrite(data_packet.data, sizeof(char), data_packet.size,
    fptr);

    bytesTransferred += data_packet.size;
    free(buffer);
    print_progress(bytesTransferred, fileCP.size);
  }
  printf("\n");
  // Wait for end control packet
  control_packet packet;
  packet.type = CONTROL_DATA;
  while (packet.type != CONTROL_END) {
    read_control_packet(fd, &packet);
  }
  printf("al: file transmission is over\n");
  update_statistics(&start_timer);
  // Close connection and cleanup
  free(fileCP.name);
  llclose(fd);
  fclose(fptr);

  return 0;
}

int read_data_packet(int fd, data_packet *packet, char *buffer)
    {
  int res = llread(fd, &buffer);
  if (res < 0) {
    al_log_msg("failed to read packet");
    return -1;
  }

  if (packet == NULL || buffer[CP_CFIELD] != CONTROL_DATA) {
    al_log_msg("unexpected control packet, aborting...");
    return -1;
  }

  packet->sequenceNr = buffer[SEQ_FIELD];
  packet->size = (uchar_t)buffer[L2_FIELD] * 256;
  packet->size += (uchar_t)buffer[L1_FIELD];
  packet->data = &buffer[L1_FIELD + 1];
  return 0;
}

```

```
273 int send_data_packet(int fd, data_packet *packet) {
274   packet->data[CP_CFIELD] = CONTROL_DATA;
275   packet->data[SEQ_FIELD] = (uchar_t)packet->sequenceNr;
276   packet->data[L2_FIELD] = (uchar_t)(packet->size / 256);
277   packet->data[L1_FIELD] = (uchar_t)(packet->size % 256);
278 
279   int res = llwrite(fd, (char *)packet->data, packet->size +
      DATA_HEADER_SIZE);
280 
281   if (res >= packet->size) return 0;
282   return -1;
283 }
284 
285 int send_control_packet(int fd, control_type type) {
286   char_buffer buffer;
287   build_control_packet(type, &buffer);
288   printf("al: sent control Packet ");
289   print_control_packet(&fileCP);
290   if (llwrite(fd, (char *)buffer.buffer, buffer.size) == -1) {
291     al_log_msg("Failed to send packet, aborting..");
292     return -1;
293   }
294   char_buffer_destroy(&buffer);
295   return 0;
296 }
297 
298 int read_control_packet(int fd, control_packet *packet) {
299   char *buffer;
300   int size = llread(fd, &buffer);
301   if (size == -1) {
302     free(buffer);
303     al_log_msg("Failed to receive packet, aborting..");
304     return -1;
305   }
306   parse_control_packet(buffer, size, packet);
307 #ifdef AL_PRINT_CPACKETS
308   printf("al: received control Packet ");
309   print_control_packet(packet);
310 #endif
311   free(buffer);
312   return 0;
313 }
314 
315 void build_control_packet(control_type type, char_buffer *
      packet) {
316   /**
317    * [C][T Size][L Size][  V Size  ][T Name][L Name][  V Name
      ]
318    * C - Control Field  T - Type  L - Length  V - Value
```

```
319     *
320     * Size: (C + 2*(T+L) = 5) + length of size + length of name
321     */
322   fileCP.type = type;
323   int packetSize = 5 + fileCP.sizeLength + fileCP.nameLength;
324   char_buffer_init(packet, packetSize);
325   char_buffer_push(packet, (char)type);
326   // SIZE TLV
327   char_buffer_push(packet, (char)TLV_SIZE_T);
328   char_buffer_push(packet, (char)fileCP.sizeLength);
329   unsigned int size = fileCP.size;
330   for (uint8_t i = 0; i < fileCP.sizeLength; ++i) {
331     char_buffer_push(packet, (char)size & 0x000000FF);
332     size >>= 8;
333   }
334   // NAME TLV
335   char_buffer_push(packet, (char)TLV_NAME_T);
336   char_buffer_push(packet, (char)fileCP.nameLength);
337   for (int i = 0; i < fileCP.nameLength; ++i) {
338     char_buffer_push(packet, fileCP.name[i]);
339   }
340 }
341
342 void print_control_packet(control_packet *packet) {
343   if (packet == NULL) return;
344
345   printf("[C %d][T 0][L %d][V %d][T 1][V %d][V %s]\n", packet->
       type,
346          packet->sizeLength, packet->size, packet->nameLength,
       packet->name);
347 }
348
349 int parse_control_packet(char *packetBuffer, int size,
       control_packet *cp) {
350   if (packetBuffer == NULL) return -1;
351   if (size < CP_MIN_SIZE) return -1;
352
353   int index = 0;
354   cp->type = packetBuffer[index++];
355
356   if (packetBuffer[index++] != TLV_SIZE_T) {
357     al_log_msg("Invalid control packet");
358     return -1;
359   }
360
361   cp->sizeLength = (int8_t)packetBuffer[index++];
362
363   cp->size = 0;
364   for (int8_t i = 0; i < cp->sizeLength; ++i) {
```

```
365     if (index > size - 1) {
366       al_log_msg("Invalid Control Packet");
367       return -1;
368     }
369     cp->size |= ((uchar_t)packetBuffer[index++]) << (8 * i);
370   }
371
372   if ((index > (size - 1)) || (packetBuffer[index++] !=
      TLV_NAME_T)) {
373     al_log_msg("Invalid control packet");
374     return -1;
375   }
376
377   if (index > (size - 1)) {
378     al_log_msg("Invalid control packet");
379     return -1;
380   }
381   cp->nameLength = (int8_t)packetBuffer[index++];
382   cp->name = (char *)malloc(cp->nameLength * sizeof(char) + 1);
383   int namePos = index;
384
385   for (int i = 0; index < (namePos + cp->nameLength); ++index)
      {
386     if (index > size) {
387       al_log_msg("Invalid control packet");
388       return -1;
389     }
390     cp->name[i++] = packetBuffer[index];
391   }
392   cp->name[cp->nameLength] = 0x00;  // Terminate string
393   return 0;
394 }
395
396 int get_file_info(const char *filename, FILE *fptr) {
397   if (fptr == NULL) return -1;
398
399   // Name
400   fileCP.name = (char *)malloc(sizeof(char) * fileCP.nameLength
        + 1);
401   strcpy(fileCP.name, filename);
402
403   // Size
404   fseek(fptr, 0L, SEEK_END);
405   fileCP.size = ftell(fptr);
406   rewind(fptr);
407
408   // Bytes needed for length
409   fileCP.sizeLength = 1;
410   int size = fileCP.size;
```

```
411   for (unsigned int i = 1; i < sizeof(int); ++i) {
412     size >>= 8;
413     if (size > 0)
414       ++fileCP.sizeLength;
415     else
416       break;
417   }
418 
419   return 0;
420 }
421 
422 void print_progress(int done, int total) {
423   float percent = ((float)done / (float)total) * 100.0f;
424   int blocks = percent / 10;
425 
426   printf("\rProgress ");
427   for (int i = 0; i < blocks; ++i) {
428     printf("#");
429   }
430   printf("[%.2f]", percent);
431   fflush(stdout);
432 }
```

link_layer.h

```
#ifndef LINK_LAYER_H
#define LINK_LAYER_H

#include "char_buffer.h"

#define LL_FLAG 0x7E     // Flag for beggining and ending of frame
#define LL_ESC 0x7D      // Escape character for byte stuffing
#define LL_ESC_MOD 0x20  // Stuffing byte
#define LL_AF1 0x03      // Transmitter commands, Receiver replys
#define LL_AF2 0x01      // Transmitter replys, Receiver commands

typedef enum {
  LL_INF = 0x00,
  LL_SET = 0x03,
  LL_DISC = 0x0B,
  LL_UA = 0x07,
  LL_RR = 0x05,
  LL_REJ = 0x01
} ll_control_type;

typedef enum {
  LL_ERROR_GENERAL = -1,
  LL_ERROR_OK = 0,
  LL_ERROR_FRAME_TOO_SMALL = -2,
  LL_ERROR_BAD_START_FLAG = -3,
  LL_ERROR_BAD_ADDRESS = -4,
  LL_ERROR_BAD_BCC1 = -5,
  LL_ERROR_BAD_END_FLAG = -6
} ll_error_code;

typedef struct {
  unsigned int frames_total;
  unsigned int frames_lost;
} ll_statistics;

/**
 * Enum of types of Link Layer connection. RECEIVER(0x01) or TRANSMITTER(0x01).
 */
typedef enum { TRANSMITTER = 0x00, RECEIVER = 0x01 } link_type;

/**
 * Closes fd and resets termios
 *
```

```
 */
void llabort(int fd);
/**
 * Sets connection settings
 *
 * @param timeout Seconds until a frame with no response times out
 * @param max_retries Number of frame retransmission attempts until failure
 * @param baudrate Serial port baudrate
 */
void ll_setup(int timeout, int max_retries, int baudrate);
/**
 * Establish connection between ports.
 *
 * @param port Number of the serial port.
 * @param type This will set the connection type to RECEIVER or TRANSMITTER
 * @return file descriptor of the connection on success, -1 on failure
 */
int llopen(int port, link_type type);
/**
 * Close connection between ports, frees memory and closes file descriptor.
 *
 * @param fd File Descriptor of an established connection
 * @return 1 on success, -1 on failure
 */
int llclose(int fd);
/**
 * Send a buffer of a given length through the connection.
 *
 * @param fd File descriptor of the connection, given by llopen.
 * @param buffer A byte array of data to transmit
 * @param length Length of the buffer
 * @return Number of bytes transmitted or -1 on failure
 */
int llwrite(int fd, char* buffer, int length);
/**
 * Read a buffer through the connection. buffer should be freed after use.
 *
 * @param fd File descriptor of the connection, given by llopen.
 * @param buffer this should be the address of a (char *), the pointer will be
 * changed to the buffer location
```

```
 * @return number of read bytes or -1 on failure
 */
int llread(int fd, char** buffer);
/**
 * Statistics of transferred bytes and accepted/rejected/
   ignored frames
 *
 * @returns a struct with link layer statistics
 */
ll_statistics ll_get_stats();

/** TESTING **/

/**
 * Builds a data frame given a buffer
 *
 * @param frame char_buffer where frame will be stored
 * @param buffer Buffer with data to frame
 * @param length Data buffer length
 */
void build_data_frame(char_buffer* frame, char* buffer, int
   length);
/**
 * Sends a buffer of raw data through the port
 *
 * @param fd File descriptor of the connection, given by llopen
   .
 * @param buffer Buffer with data to send
 * @param length Data buffer length
 */
int send_raw_data(int fd, char* buffer, int length);
/**
 * Initiates serial port connection
 *
 * @param port Number of the serial port.
 * @param type This will set the connection type to RECEIVER or
    TRANSMITTER
 * @return Number of bytes written, -1 on failure
 */
int init_serial_port(int port, link_type type);
/**
 * Closes serial port connection
 *
 *  @param fd File descriptor of the connection, given by
   llopen.
 */
void close_serial_port(int fd);

#endif
```

link_layer.c

```
1
2  #include "link_layer.h"
3
4  #include <sys/types.h>
5  #include <sys/stat.h>
6  #include <fcntl.h>
7  #include <termios.h>
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include <unistd.h>
11 #include <string.h>
12 #include <stdbool.h>
13 #include <signal.h>
14
15 //#define LL_LOG_INFORMATION  // Log general information
16 //#define LL_LOG_BUFFER       // Log entire frame
17 //#define LL_LOG_FRAMES  // Log frame headers
18
19 /*FER Testing */
20 //#define FER
21 #define FERPROB 20
22
23 /* POSIX compliant source */
24 #define _POSIX_SOURCE 1
25
26 #define INF_FRAME_SIZE 6
27 #define CONTROL_FRAME_SIZE 5
28 #define MAX_PORT_LENGTH 20
29
30 #define MAX_TRANSMISSION_ATTEMPS 3
31 #define TIMEOUT_DURATION 3
32 #define DEFAULT_BAUDRATE B38400
33
34 /*  Port name prefix */
35 #ifdef __linux__
36 #define PORT_NAME "/dev/ttyS"
37 #elif _WIN32
38 #define PORT_NAME "COM"
39 #else
40 #define PORT_NAME "/dev/ttyS"
41 #endif
42
43 typedef unsigned char uchar_t;
44 typedef struct {
45   char port[MAX_PORT_LENGTH];
46   int baud_rate;
47   unsigned int sequence_number;
```

```
48   unsigned int timeout;
49   unsigned int num_transmissions;
50 } link_layer;
51
52 typedef enum {
53   FD_FIELD = 0x00,    // Start Flag
54   AF_FIELD = 0x01,    // Address
55   C_FIELD = 0x02,     // Control
56   BCC1_FIELD = 0x03,  // BCC1
57   DATA_FIELD = 0x04   // Data field start
58 } ll_control_frame_field;
59
60 struct termios oldtio;
61 static link_layer ll;
62 static link_type ltype;
63 static ll_statistics stats;
64 static bool ll_init = false;
65 static int afd;
66
67 volatile bool alarm_triggered;
68 volatile unsigned int transmission_attempts = 0;
69
70 /* Declarations */
71 // Link Layer
72 int ll_open_transmitter(int fd);
73 int ll_open_receiver(int fd);
74 int read_frame(int fd, char_buffer *frame);
75 int validate_control_frame(char_buffer *frame);
76 void build_control_frame(char_buffer *frame, ll_control_type
       type);
77 void build_data_frame(char_buffer *frame, char *buffer, int
       length);
78 char get_address_field(link_type lnk, ll_control_type type);
79 bool is_control_command(ll_control_type type);
80 bool is_frame_control_type(char_buffer *frame, ll_control_type
       type);
81 void printControlType(ll_control_type type);
82 int send_frame(int fd, char_buffer *frame);
83 int frame_exchange(int fd, char_buffer *frame, ll_control_type
       reply);
84 int send_control_frame(int fd, ll_control_type type);
85 void log_frame(char_buffer *frame, const char *type);
86 int get_termios_baudrate(int baudrate);
87
88 void log_msg(const char *msg) {
89 #ifdef LL_LOG_INFORMATION
90   fprintf(stderr, "ll: %s\n", msg);
91 #endif
92 }
```

```
93
94  /**
95   *
96   *
97   * SIGNALS
98   *
99   */
100
101 void sig_alarm_handler(int sig_num) {
102   if (sig_num == SIGALRM) {
103     ++transmission_attempts;
104     alarm_triggered = true;
105   }
106 }
107 void set_alarm(unsigned int seconds) {
108   alarm_triggered = false;
109   alarm(seconds);
110 }
111
112 void set_alarm_handler() {
113   signal(SIGALRM, sig_alarm_handler);
114   alarm_triggered = false;
115   transmission_attempts = 0;
116 }
117
118 void reset_alarm_handler() {
119   alarm(0);
120   signal(SIGALRM, NULL);
121   alarm_triggered = false;
122   transmission_attempts = 0;
123 }
124
125 bool was_alarm_triggered() {
126   if (alarm_triggered) {
127     log_msg("connection timed out...");
128     return true;
129   }
130   return false;
131 }
132
133 // Reset termios on CTRL+C
134 void sig_int_handler(int sig) {
135   if (sig == SIGINT) {
136     tcsetattr(afd, TCSANOW, &oldtio);
137     exit(-1);
138   }
139 }
140
141 /**
```

```
 *
 *
 * LINK LAYER
 *
 */

void llabort(int fd) { close_serial_port(fd); }

void ll_setup(int timeout, int max_retries, int baudrate) {
  ll.timeout = timeout;
  ll.num_transmissions = max_retries;
  ll.baud_rate = get_termios_baudrate(baudrate);
  ll_init = true;
}

// LLOPEN
int llopen(int port, link_type type) {
  signal(SIGINT, sig_int_handler);
  afd = init_serial_port(port, type);
  if (afd == -1) return -1;

  if (type == TRANSMITTER) return ll_open_transmitter(afd);
  return ll_open_receiver(afd);
}

// LLCLOSE
int llclose(int fd) {
  log_msg("llclose - communicating disconnect\n");

  if (ltype == TRANSMITTER) {
    // Send Disc, receive DISC
    char_buffer discFrame;
    build_control_frame(&discFrame, LL_DISC);
    if (frame_exchange(fd, &discFrame, LL_DISC) == -1) {
      log_msg("warning - failed to communicate disconnect");
      char_buffer_destroy(&discFrame);
      close_serial_port(fd);
      return LL_ERROR_GENERAL;
    }

    char_buffer_destroy(&discFrame);

    // Send UA
    send_control_frame(fd, LL_UA);
    usleep(50);
    log_msg("llclose - disconnected.\n");
    close_serial_port(fd);
    return 1;
  }
```

```
191
192   if (ltype == RECEIVER) {
193     while (true) {
194       /* Wait for DISC */
195       char_buffer reply_frame;
196       int res = read_frame(fd, &reply_frame);
197       bool is_disc = is_frame_control_type(&reply_frame,
    LL_DISC);
198       char_buffer_destroy(&reply_frame);
199
200       if (!is_disc) log_msg("frame ignored - unexpected control
     field");
201
202       // If invalid frame or not a DISC command, retry
203       if (res != -1 && is_disc) break;
204     }
205     /* Send DISC, receive UA */
206     char_buffer disc_frame;
207     build_control_frame(&disc_frame, LL_DISC);
208     int res = frame_exchange(fd, &disc_frame, LL_UA);
209     if (res == -1) log_msg("llclose - failed to communicate
    disconnect");
210
211     log_msg("llclose - disconnected.\n");
212     close_serial_port(fd);
213     return res;
214   }
215   close_serial_port(fd);
216   return LL_ERROR_GENERAL;
217 }
218
219 // LLWRITE
220 int llwrite(int fd, char *buffer, int length) {
221   char_buffer frame;
222   build_data_frame(&frame, buffer, length);
223   if (frame_exchange(fd, &frame, LL_RR) == -1) {
224     log_msg("llwrite failed");
225
226     char_buffer_destroy(&frame);
227     return -1;
228   }
229
230   int size = frame.size;
231   ll.sequence_number ^= 1;
232   char_buffer_destroy(&frame);
233   return size;
234 }
235
236 // LLREAD
```

```
237 int llread(int fd, char **buffer) {
238   char_buffer frame;
239   while (true) {
240     if (read_frame(fd, &frame) == -1) {
241       char_buffer_destroy(&frame);
242       continue;
243     }
244 
245     // Ignore
246     if (!is_frame_control_type(&frame, LL_INF)) {
247       if (is_frame_control_type(&frame, LL_SET)) {
248         log_msg("frame ignored - unexpected SET control");
249         send_control_frame(fd, LL_UA);
250       } else
251         log_msg("frame ignored - unexpected control field");
252       char_buffer_destroy(&frame);
253       continue;
254     }
255 
256     // Check seq number for duplicate frames
257     if ((frame.buffer[C_FIELD] >> 6) == (uchar_t)ll.
    sequence_number) {
258       log_msg("frame ignored - duplicate");
259       send_control_frame(fd, LL_RR);
260       continue;
261     }
262 
263     char_buffer packet;
264     char_buffer_init(&packet, INF_FRAME_SIZE);
265 
266     uchar_t bcc2 = 0x00;  // Calculated BCC2
267     // Get the packet BBC2 value, check for ESC_MOD
268     uchar_t packet_bcc2 = frame.buffer[frame.size - 2];
269     unsigned int dataLimit = frame.size - 2;
270 
271     // Adjust for BCC2 escape flag
272     if (frame.buffer[frame.size - 3] == LL_ESC) {
273       packet_bcc2 ^= LL_ESC_MOD;
274       --dataLimit;
275 #ifdef DEBUG_PRINT_INFORMATION
276       printf("BCC2 after byte destuffing: %x\n", packet_bcc2);
277 #endif
278     }
279     // Destuff bytes and calculate BCC2
280     for (unsigned int i = DATA_FIELD; i < dataLimit; ++i) {
281       uchar_t temp;
282       if (frame.buffer[i] == LL_ESC) {
283         temp = (uchar_t)(frame.buffer[++i]) ^ LL_ESC_MOD;
284 
```

```
285       } else
286         temp = (uchar_t)frame.buffer[i];
287
288       bcc2 ^= temp;
289       char_buffer_push(&packet, temp);
290     }
291
292     // BCC2 check
293     if (bcc2 != packet_bcc2) {
294       printf(
295           "ll: frame rejected - failed BBC2 check, expected: %x
    , received %x\n",
296           bcc2, packet_bcc2);
297       send_control_frame(fd, LL_REJ);
298       char_buffer_destroy(&packet);
299       char_buffer_destroy(&frame);
300       continue;
301     }
302
303 #ifdef FER
304     unsigned int randomN = rand() % 100;
305     if (randomN < FERPROB) {
306       send_control_frame(fd, LL_REJ);
307       char_buffer_destroy(&packet);
308       char_buffer_destroy(&frame);
309       continue;
310     }
311 #endif
312     // Frame read successfuly, flip seq number and reply with
    RR
313     ll.sequence_number ^= 1;
314     send_control_frame(fd, LL_RR);
315     char_buffer_destroy(&frame);
316     *buffer = (char *)packet.buffer;
317     return packet.size;
318   }
319   return LL_ERROR_GENERAL;
320 }
321
322 ll_statistics ll_get_stats() { return stats; }
323
324 /**
325  *
326  *
327  * LINK LAYER AUXILIAR
328  *
329  */
330
331 int ll_open_transmitter(int fd) {
```

```
332   ll.sequence_number = 1;
333
334   char_buffer set_frame;
335   build_control_frame(&set_frame, LL_SET);
336   if (frame_exchange(fd, &set_frame, LL_UA) == -1) {
337     char_buffer_destroy(&set_frame);
338     return LL_ERROR_GENERAL;
339   }
340   char_buffer_destroy(&set_frame);
341
342   log_msg("llopen - Connected.\n");
343   return fd;
344 }
345
346 int ll_open_receiver(int fd) {
347   ll.sequence_number = 0;
348   while (true) {
349     log_msg("llopen - Wainting for connection\n");
350
351     ll_control_type type = LL_DISC;
352     while (type != LL_SET) {
353       char_buffer set_frame;
354       if (read_frame(fd, &set_frame) == -1) continue;
355
356       type = set_frame.buffer[C_FIELD];
357
358       if (type != LL_SET) log_msg("frame ignored - unexpected
    control field");
359
360       char_buffer_destroy(&set_frame);
361     }
362     if (send_control_frame(fd, LL_UA) == -1) {
363       log_msg("llopen - was unable to send UA packet");
364       return LL_ERROR_GENERAL;
365     }
366
367     log_msg("llopen - connected.\n");
368     return fd;
369   }
370 }
371
372 /*
373  *
374  *
375  * FRAMES
376  *
377  */
378
379 int frame_exchange(int fd, char_buffer *frame, ll_control_type
```

```
   reply) {
380   set_alarm_handler();
381   while (transmission_attempts < ll.num_transmissions) {
382     send_frame(fd, frame);
383     set_alarm(ll.timeout);
384     while (true) {
385       char_buffer reply_frame;
386       int res = read_frame(fd, &reply_frame);
387
388       // Timeout or invalid frame received
389       if (res == -1) {
390         tcflush(fd, TCIOFLUSH);
391         char_buffer_destroy(&reply_frame);
392         break;
393       }
394
395       bool is_rej = is_frame_control_type(&reply_frame, LL_REJ)
    ;
396       bool is_rr = is_frame_control_type(&reply_frame, LL_RR);
397
398       // Verify if RR is a duplicate
399       if (is_rr) {
400         uchar_t replySeq = ((uchar_t)(reply_frame.buffer[
    C_FIELD]) >> 7);
401         if (replySeq != (uchar_t)(ll.sequence_number)) {
402           log_msg("frame ignored - duplicate");
403           ++stats.frames_lost;
404           char_buffer_destroy(&reply_frame);
405           continue;
406         }
407       }
408       // Received REJ, resend frame if not duplicate
409       if (is_rej) {
410         uchar_t replySeq = ((uchar_t)(reply_frame.buffer[
    C_FIELD]) >> 7);
411         if (replySeq != (uchar_t)(ll.sequence_number ^ 1)) {
412           log_msg("frame ignored - duplicate");
413           ++stats.frames_lost;
414           char_buffer_destroy(&reply_frame);
415           continue;
416         }
417         char_buffer_destroy(&reply_frame);
418         log_msg("frame Rejected by receiver");
419         ++stats.frames_lost;
420         ++transmission_attempts;
421         break;
422       }
423       // Exchange was sucessful
424       if (is_frame_control_type(&reply_frame, reply)) {
```

```
425         char_buffer_destroy(&reply_frame);
426         reset_alarm_handler();
427         return 1;
428       } else {
429         log_msg("frame ignored - unexpected control field");
430         ++stats.frames_lost;
431       }
432     }
433   }
434
435   reset_alarm_handler();
436   log_msg("error: exceeded transmission attempts, connection
      failed");
437
438   return LL_ERROR_GENERAL;
439 }
440
441 int send_control_frame(int fd, ll_control_type type) {
442   char_buffer frame;
443   build_control_frame(&frame, type);
444   int res = send_frame(fd, &frame);
445   char_buffer_destroy(&frame);
446   return res;
447 }
448
449 int send_frame(int fd, char_buffer *frame) {
450   if (write(fd, frame->buffer, frame->size) < 0) {
451     log_msg("warning - unable to write frame to port");
452     return LL_ERROR_GENERAL;
453   }
454   log_frame(frame, "sent");
455   return 0;
456 }
457
458 int read_frame(int fd, char_buffer *frame) {
459   if (frame == NULL) return LL_ERROR_GENERAL;
460   char_buffer_init(frame, CONTROL_FRAME_SIZE);
461
462   char inc_byte = 0x00;
463   int read_status = 0;
464   // Clear buffer and wait for a flag
465   while (read_status <= 0 ||
466          (uchar_t)inc_byte != LL_FLAG) {  // TO-DO Implement
      timeout
467     if (was_alarm_triggered()) return LL_ERROR_GENERAL;
468     read_status = read(fd, &inc_byte, 1);
469   }
470   char_buffer_push(frame, (uchar_t)inc_byte);
471   // Reset vars
```

```
472   read_status = 0;
473   inc_byte = 0x00;
474   // Read serial until flag is found
475   while (inc_byte != LL_FLAG) {
476     if (was_alarm_triggered()) return LL_ERROR_GENERAL;
477     read_status = read(fd, &inc_byte, 1);
478     if (read_status <= 0) continue;
479     char_buffer_push(frame, inc_byte);
480   }
481
482   ++stats.frames_total;
483
484 #ifdef LL_LOG_BUFFER
485   char_buffer_printHex(frame);
486 #endif
487
488   if (validate_control_frame(frame) < 0) {
489     log_msg("frame ignored - failed validation of header");
490     ++stats.frames_lost;
491     return LL_ERROR_GENERAL;
492   }
493
494   return LL_ERROR_OK;
495 }
496
497 int validate_control_frame(char_buffer *frame) {
498   if (frame == NULL || frame->buffer == NULL) return -1;
499
500   if (frame->size < CONTROL_FRAME_SIZE) {
501     char_buffer_printHex(frame);
502     return LL_ERROR_FRAME_TOO_SMALL;
503   }
504
505 #ifdef LL_LOG_FRAMES
506   log_frame(frame, "received");
507 #endif
508
509   // Start Flag
510   if (frame->buffer[FD_FIELD] != LL_FLAG) return
      LL_ERROR_BAD_START_FLAG;
511   // Check address
512   char expectedAF = get_address_field(ltype ^ 1, frame->buffer[
      C_FIELD]);
513   if (frame->buffer[AF_FIELD] != expectedAF) return
      LL_ERROR_BAD_ADDRESS;
514   // Check BCC1
515   if (frame->buffer[BCC1_FIELD] !=
516       (uchar_t)(expectedAF ^ frame->buffer[C_FIELD]))
517     return LL_ERROR_BAD_BCC1;
```

```
518   // Last element flag
519   if (frame->buffer[frame->size - 1] != (uchar_t)LL_FLAG)
520     return LL_ERROR_BAD_END_FLAG;
521
522   return LL_ERROR_OK;
523 }
524
525 void build_control_frame(char_buffer *frame, ll_control_type
      type) {
526   char_buffer_init(frame, CONTROL_FRAME_SIZE);
527   char_buffer_push(frame, LL_FLAG);                          //
      FLAG
528   char_buffer_push(frame, get_address_field(ltype, type));  //
      ADDRESS
529
530   if (type == LL_RR || type == LL_REJ) type |= ll.
      sequence_number << 7;  // N(r)
531
532   char_buffer_push(frame, type);
        // Control type
533   char_buffer_push(frame, frame->buffer[1] ^ frame->buffer[2]);
        // BCC1
534   char_buffer_push(frame, LL_FLAG);
        // FLAG
535 }
536
537 void build_data_frame(char_buffer *frame, char *buffer, int
      length) {
538   char_buffer_init(frame, length + INF_FRAME_SIZE);
539   char_buffer_push(frame, LL_FLAG);
      // FLAG
540   char_buffer_push(frame, get_address_field(ltype, LL_INF));
      // ADDRESS
541   char_buffer_push(frame,
542                    LL_INF | (ll.sequence_number << 6));  //
      Control and N(s)
543   char_buffer_push(frame, frame->buffer[1] ^ frame->buffer[2]);
        // BCC1
544
545   // Add buffer to frame and calculate bcc2
546   uchar_t bcc2 = 0x00;
547   for (int i = 0; i < length; ++i) {
548     // BCC2
549     bcc2 ^= (uchar_t)buffer[i];
550     // Byte stuffing
551     if (buffer[i] == (uchar_t)LL_FLAG || buffer[i] == (uchar_t)
      LL_ESC) {
552       char_buffer_push(frame, (uchar_t)LL_ESC);
553       char_buffer_push(frame, buffer[i] ^ (uchar_t)LL_ESC_MOD);
```

```
554     } else
555       char_buffer_push(frame, buffer[i]);
556   }
557
558   // Bytestuffin on BBC2 when needed
559   if (bcc2 == (uchar_t)LL_ESC || bcc2 == (uchar_t)LL_FLAG) {
560     char_buffer_push(frame, (uchar_t)LL_ESC);
561     char_buffer_push(frame, (uchar_t)(bcc2 ^ (uchar_t)
      LL_ESC_MOD));
562   } else
563     char_buffer_push(frame, bcc2);
564
565   char_buffer_push(frame, (uchar_t)LL_FLAG);
566
567 #ifdef LL_LOG_BUFFER
568   char_buffer_printHex(frame);
569 #endif
570 }
571
572 char get_address_field(link_type lnk, ll_control_type type) {
573   type &= 0x0F;
574   if (lnk == RECEIVER && is_control_command(type))
575     return LL_AF2;
576   else if (lnk == TRANSMITTER && !is_control_command(type))
577     return LL_AF2;
578   return LL_AF1;
579 }
580
581 bool is_control_command(ll_control_type type) {
582   type &= 0x0F;
583   if (type == LL_INF || type == LL_DISC || type == LL_SET)
      return true;
584   return false;
585 }
586
587 bool is_frame_control_type(char_buffer *frame, ll_control_type
      type) {
588   if (frame == NULL || frame->buffer == NULL) return false;
589   type &= 0x0F;
590   ll_control_type frameType = frame->buffer[C_FIELD] & 0x0F;
591   return frameType == type;
592 }
593
594 const char *get_control_type_str(ll_control_type type) {
595   type &= 0x0F;
596   switch (type) {
597     case LL_INF: {
598       return "INF";
599     }
```

```
600     case LL_SET: {
601       return "SET";
602     }
603     case LL_DISC: {
604       return "DISC";
605     }
606     case LL_UA: {
607       return "UA";
608     }
609     case LL_RR: {
610       return "RR";
611     }
612     case LL_REJ: {
613       return "REJ";
614     }
615     default:
616       return "UNK";
617   }
618 }
619
620 void log_frame(char_buffer *frame, const char *type) {
621 #ifdef LL_LOG_FRAMES
622   printf("ll: %s packet %s", type,
623          get_control_type_str(frame->buffer[C_FIELD]));
624
625   // Header
626   printf("\t\t\t[F %x][A %x][C %x][BCC1 %x]", frame->buffer[
      FD_FIELD],
627          frame->buffer[AF_FIELD], (uchar_t)frame->buffer[
      C_FIELD],
628          (uchar_t)frame->buffer[BCC1_FIELD]);
629   // BCC2
630   if ((frame->buffer[C_FIELD] & 0x0F) == LL_INF)
631     printf("[BCC2 %x]", (uchar_t)frame->buffer[frame->size -
      2]);
632
633   // Tail
634   printf("[F %x] Frame size: %d bytes\n", frame->buffer[frame->
      size - 1],
635          frame->size);
636 #endif
637 }
638
639 /**
640  *
641  *
642  * SERIAL PORT
643  *
644  */
```

```
645
646 int get_termios_baudrate(int baudrate) {
647   switch (baudrate) {
648     case 0:
649       return B0;
650     case 50:
651       return B50;
652     case 75:
653       return B75;
654     case 110:
655       return B110;
656     case 134:
657       return B134;
658     case 150:
659       return B150;
660     case 200:
661       return B200;
662     case 300:
663       return B300;
664     case 600:
665       return B600;
666     case 1200:
667       return B1200;
668     case 1800:
669       return B1800;
670     case 2400:
671       return B2400;
672     case 4800:
673       return B4800;
674     case 9600:
675       return B9600;
676     case 19200:
677       return B19200;
678     case 38400:
679       return B38400;
680     case 57600:
681       return B57600;
682     case 115200:
683       return B115200;
684     case 230400:
685       return B230400;
686     case 460800:
687       return B460800;
688     default:
689       return DEFAULT_BAUDRATE;
690   }
691 }
692
693 int init_serial_port(int port, link_type type) {
```

```
694   // Init ll struct
695   snprintf(ll.port, MAX_PORT_LENGTH, "%s%d", PORT_NAME, port);
696   if (!ll_init)
697     ll_setup(TIMEOUT_DURATION, MAX_TRANSMISSION_ATTEMPS,
      DEFAULT_BAUDRATE);
698
699   ltype = type;
700
701   int fd = open(ll.port, O_RDWR | O_NOCTTY);
702   if (fd < 0) {
703     perror(ll.port);
704     return -1;
705   }
706
707   struct termios newtio;
708
709   if (tcgetattr(fd, &oldtio) == -1) { /* save current port
      settings */
710     perror("tcgetattr");
711     close(fd);
712     return -1;
713   }
714
715   bzero(&newtio, sizeof(newtio));
716   newtio.c_cflag = ll.baud_rate | CS8 | CLOCAL | CREAD;
717   newtio.c_iflag = IGNPAR;
718   newtio.c_oflag = 0;
719
720   /* set input mode (non-canonical, no echo,...) */
721   newtio.c_lflag = 0;
722
723   newtio.c_cc[VTIME] = 20; /* inter-character timer unused */
724   newtio.c_cc[VMIN] = 0;   /* blocking read until 5 chars
      received */
725
726   /*
727     VTIME e VMIN devem ser alterados de forma a proteger com um
       temporizador a
728     leitura do(s)  p r ximo (s) caracter(es)
729   */
730
731   tcflush(fd, TCIOFLUSH);
732
733   if (tcsetattr(fd, TCSANOW, &newtio) == -1) {
734     perror("tcsetattr");
735     close(fd);
736     return -1;
737   }
738
```

```
  return fd;
}

void close_serial_port(int fd) {
  tcsetattr(fd, TCSANOW, &oldtio);
  close(fd);
}

/** Testing **/
int send_raw_data(int fd, char *buffer, int length) {
  return write(fd, buffer, length);
}
```

char_buffer.h

```
#ifndef CHAR_BUFFER_H
#define CHAR_BUFFER_H

typedef struct {
  unsigned char *buffer;
  unsigned int size;
  unsigned int capacity;
} char_buffer;

int char_buffer_init(char_buffer *cb, int initSize);
int char_buffer_push(char_buffer *cb, char bt);
int char_buffer_remove(char_buffer *cb, unsigned int pos);
void char_buffer_print(char_buffer *cb);
void char_buffer_printHex(char_buffer *cb);
void char_buffer_destroy(char_buffer *cb);

#endif
```

char_buffer.c

```c
#include "char_buffer.h"

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

int char_buffer_init(char_buffer *cb, int initSize) {
  if (cb == NULL) return -1;

  cb->buffer = (unsigned char *)malloc(initSize * sizeof(
    unsigned char));
  if (cb->buffer == NULL) return -1;
  cb->size = 0;
  cb->capacity = initSize * sizeof(unsigned char);
  return 0;
}

int char_buffer_push(char_buffer *cb, char bt) {
  if (cb == NULL || cb->buffer == NULL) return -1;

  // Grow if at capacity
  if (cb->size * sizeof(unsigned char) >= cb->capacity) {
    cb->buffer = (unsigned char *)realloc(cb->buffer, cb->
    capacity * 2);
    if (cb->buffer == NULL) return -1;
    cb->capacity *= 2;
  }

  cb->buffer[cb->size] = bt;
  ++cb->size;

  return 0;
}

int char_buffer_remove(char_buffer *cb, unsigned int pos) {
  if (cb == NULL || cb->buffer == NULL || pos > cb->size + 1)
    return -1;

  // Pop last element
  if (pos == cb->size + 1) {
    cb->buffer[pos] = (unsigned char)0x00;
    --cb->size;
    return 0;
  }

  memmove(cb->buffer + pos * sizeof(unsigned char),
          cb->buffer + (pos + 1) * sizeof(unsigned char),
```

```
45            cb->capacity - (pos + 1) * sizeof(unsigned char));
46
47    --cb->size;
48    return 0;
49  }
50
51  void char_buffer_print(char_buffer *cb) {
52    if (cb == NULL || cb->buffer == NULL) return;
53    printf("Buffer Content: ");
54    for (unsigned int i = 0; i < cb->size; ++i) printf("%c ", cb
        ->buffer[i]);
55    printf(" Elements: %d Capacity: %d", cb->size, cb->capacity);
56    printf("\n\n");
57  }
58
59  void char_buffer_printHex(char_buffer *cb) {
60    if (cb == NULL || cb->buffer == NULL) return;
61    printf("Buffer Content: ");
62    for (unsigned int i = 0; i < cb->size; ++i) printf("%x ", cb
        ->buffer[i]);
63    printf(" Elements: %d Capacity: %d", cb->size, cb->capacity);
64    printf("\n\n");
65  }
66
67  void char_buffer_destroy(char_buffer *cb) {
68    if (cb == NULL || cb->buffer == NULL) return;
69    free(cb->buffer);
70  }
```