

# Redes de Computadores

## 1.º Trabalho Laboratorial

Diogo Costa  
up201906731@edu.fe.up.pt

Francisco Colino  
up201905405@edu.fe.up.pt

4 de dezembro de 2021

### Sumário

- 1 Introdução**
- 2 Arquitetura**
- 3 Estrutura do código**
- 4 Casos de uso principais**
- 5 Protocolo de ligação lógica**
- 6 Protocolo de aplicação**
- 7 Validação**
- 8 Eficiência do protocolo de ligação de dados**
- 9 Conclusões**

## 10 Anexos

### 10.1 Código Fonte

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4
5 #include "aplic.h"
6
7
8 int main(int argc, char** argv) {
9     if (argc != 4) {
10         printf("Usage:\tsender SerialPort Path NameToGive\n\tex: nserial <i> <path> <name>\n");
11         return -1;
12     }
13
14     int porta = atoi(argv[1]);
15
16     if (send_file(porta, argv[2], strlen(argv[2]), argv[3]) < 0) {
17         return -1;
18     }
19
20     return 0;
21 }
```

Listing 1: sender.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #include "aplic.h"
5
6
7 int main(int argc, char** argv) {
8     if (argc != 2) {
9         printf("Usage:\treceiver SerialPort\n\tex: nserial <i>\n");
10         return -1;
11     }
12
13     int porta = atoi(argv[1]);
14
15     if (receive_file(porta) < 0 ) {
16         return -1;
17     }
18
19     return 0;
20 }
```

Listing 2: receiver.c

```

1 int send_file(int porta, char *path, int path_size, char *
    file_name);
2
3 int receive_file(int porta);

```

Listing 3: aplic.h

```

1 #include <stdint.h>
2
3 #define BAUDRATE B38400
4 #define _POSIX_SOURCE 1 /* POSIX compliant source */
5 #define FALSE 0
6 #define TRUE 1
7
8 #define DATA_PACKET_MAX_SIZE 1000
9
10 typedef enum type {
11     TRANSMITTER,
12     RECEIVER
13 } type_t;
14
15 int llopen(int porta, type_t type);
16
17 int llclose(int fd, type_t type);
18
19 int llwrite(int fd, uint8_t *buffer, int length);
20
21 int llread(int fd, uint8_t *buffer);

```

Listing 4: linklayer.h

```

1 #include "aplic.h"
2 #include "linklayer.h"
3 #include <sys/types.h>
4 #include <sys/stat.h>
5 #include <fcntl.h>
6 #include <termios.h>
7 #include <stdio.h>
8 #include <string.h>
9 #include <strings.h>
10 #include <stdlib.h>
11 #include <unistd.h>
12 #include <stdint.h>
13
14 #define CONTROL_PACKET_MAX_SIZE 500
15 #define PACKET_MAX_SIZE (CONTROL_PACKET_MAX_SIZE >
    DATA_PACKET_MAX_SIZE ? CONTROL_PACKET_MAX_SIZE :
    DATA_PACKET_MAX_SIZE)
16 #define FILE_NAME_MAX_SIZE 255
17
18 #define C_DATA 0x1
19 #define C_START 0x2
20 #define C_END 0x3
21

```

```

22 #define N(seq) ((seq) % 255)
23
24 #define L1(K) ((K) & 0b11111111)
25 #define L2(K) (((K) >> 8) & 0b11111111)
26 #define K(L1,L2) (256*(L2)+(L1))
27
28 #define T_FILE_SIZE 0x0
29 #define T_FILE_NAME 0x1
30
31
32 static off_t get_file_size(int fd) {
33     struct stat s;
34     if (fstat(fd, &s) == -1) {
35         return -1;
36     }
37
38     return s.st_size;
39 }
40
41 static uint8_t* get_control_packet(off_t file_size, char *
    file_name, int file_name_size, int *length) {
42     uint8_t* control_packet = malloc(CONTROL_PACKET_MAX_SIZE);
43     if (control_packet == NULL) {
44         return NULL;
45     }
46
47     size_t i = 0;
48
49     control_packet[i++] = C_START;
50     control_packet[i++] = T_FILE_SIZE; // T1
51     control_packet[i++] = sizeof(off_t); // L1
52
53     memcpy(&control_packet[i], &file_size, sizeof(off_t)); //
V1
54     i += sizeof(off_t);
55
56     control_packet[i++] = T_FILE_NAME; // T2
57
58     control_packet[i++] = (uint8_t)file_name_size; // L2
59
60     memcpy(&control_packet[i], file_name, file_name_size); //
V2
61
62     *length = i + file_name_size;
63     return control_packet;
64 }
65
66 static int send_packaged_file(int fd_serial_port, int fd_file)
67 {
68     uint8_t *data_packet = malloc(DATA_PACKET_MAX_SIZE);
69     if (data_packet == NULL) {
70         return -1;
71     }

```

```

72     uint8_t sequence_number = 0;
73     data_packet[0] = C_DATA;
74
75     while (1) {
76         data_packet[1] = sequence_number;
77         sequence_number = (sequence_number+1) % 255;
78
79         ssize_t num = read(fd_file, &data_packet[4],
80 DATA_PACKET_MAX_SIZE-4);
81
82         if (num == -1) {
83             free(data_packet);
84             return -1;
85         } else if (num == 0) {
86             break;
87         } else {
88             data_packet[2] = L2(num);
89             data_packet[3] = L1(num);
90
91             if (llwrite(fd_serial_port, data_packet, num+4) <
92 0) {
93                 free(data_packet);
94                 return -1;
95             }
96         }
97
98         free(data_packet);
99         return 0;
100 }
101
102 int send_file(int porta, char *path, int path_size, char *
103 file_name) {
104     int file_name_size = strlen(file_name);
105     if (file_name_size > FILE_NAME_MAX_SIZE) {
106         printf("File name too big.\n");
107         return -1;
108     }
109
110     int fd_file;
111     if ((fd_file = open(path, O_RDONLY)) < 0) {
112         printf("File not found.\n");
113         return -1;
114     }
115
116     off_t file_size = 0;
117     if ((file_size = get_file_size(fd_file)) < 0) {
118         close(fd_file);
119         return -1;
120     }
121
122     int fd_serial_port;
123     if ((fd_serial_port = llopen(porta, TRANSMITTER)) < 0) {
124         close(fd_file);

```

```

123         return -1;
124     }
125
126     uint8_t* control_packet = NULL;
127     int control_packet_size = 0;
128     if ((control_packet = get_control_packet(file_size,
129     file_name, file_name_size, &control_packet_size)) == NULL)
130     {
131         close(fd_file);
132         llclose(fd_serial_port, TRANSMITTER);
133         return -1;
134     }
135
136     // Control packet start
137     if (llwrite(fd_serial_port, control_packet,
138     control_packet_size) < 0) {
139         free(control_packet);
140         close(fd_file);
141         llclose(fd_serial_port, TRANSMITTER);
142         return -1;
143     }
144
145     if (send_packaged_file(fd_serial_port, fd_file) != 0) {
146         free(control_packet);
147         close(fd_file);
148         llclose(fd_serial_port, TRANSMITTER);
149         return -1;
150     }
151
152     // Control packet end
153     control_packet[0] = C_END;
154     if (llwrite(fd_serial_port, control_packet,
155     control_packet_size) < 0) {
156         free(control_packet);
157         close(fd_file);
158         llclose(fd_serial_port, TRANSMITTER);
159         return -1;
160     }
161
162     free(control_packet);
163     close(fd_file);
164     llclose(fd_serial_port, TRANSMITTER);
165     return 0;
166 }
167
168 int receive_file(int porta) {
169     int fd_serial_port;
170     if ((fd_serial_port = llopen(porta, RECEIVER)) < 0) {
171         return -1;
172     }
173
174     uint8_t *packet = malloc(PACKET_MAX_SIZE);
175     if (packet == NULL) {
176         llclose(fd_serial_port, RECEIVER);

```

```

173         return -1;
174     }
175
176     int fd_file_to_write = -1;
177     int sequence_number = 0;
178     off_t file_size = 0;
179
180     int not_end_packet = TRUE;
181     while (not_end_packet) {
182         int packet_size = 0;
183         if ((packet_size = llread(fd_serial_port, packet)) < 0)
184         {
185             free(packet);
186             llclose(fd_serial_port, RECEIVER);
187             return -1;
188         }
189
190         uint8_t control_field = packet[0];
191         switch (control_field) {
192             case C_DATA:
193                 if (fd_file_to_write == -1) { // Control start
194                     packet didn't arrive yet
195                     break;
196                 }
197
198                 int N = packet[1];
199
200                 if (N == sequence_number) {
201                     int num_octets = K(packet[3], packet[2]);
202                     if (write(fd_file_to_write, &packet[4],
203 num_octets) == -1) {
204                         free(packet);
205                         llclose(fd_serial_port, RECEIVER);
206                         return -1;
207                     }
208                     sequence_number = (sequence_number + 1) % 255;
209                 } else {
210                     // wrong sequence number
211                     printf("Wrong packet sequence number. Expected:
212 %d ; Got: %d\n", sequence_number, N);
213                     free(packet);
214                     llclose(fd_serial_port, RECEIVER);
215                     return -1;
216                 }
217
218                 break;
219
220             case C_START:;
221                 int i = 1;
222                 while (i < packet_size) {
223                     uint8_t T = packet[i++];
224                     uint8_t L = packet[i++];
225
226                     if (T == T_FILE_SIZE) {

```

```

223         memcpy(&file_size, &packet[i], L);
224
225     } else if (T == T_FILE_NAME) {
226         char file_name[FILE_NAME_MAX_SIZE];
227         memcpy(file_name, &packet[i], L);
228
229         fd_file_to_write = open(file_name, O_WRONLY
230 | O_APPEND | O_CREAT, 0644);
231         if (fd_file_to_write == -1) {
232             free(packet);
233             llclose(fd_serial_port, RECEIVER);
234             return -1;
235         }
236     } else {
237         printf("ERROR: not supposed to reach this\n
238 ");
239     }
240     i += L;
241 }
242
243 break;
244
245 case C_END:
246     if (fd_file_to_write != -1) {
247         close(fd_file_to_write);
248         not_end_packet = FALSE;
249     }
250     // else Control start packet didn't arrive yet, so
251     wait for it
252
253     break;
254
255 default:
256     break; // invalid control field (ignore packet)
257 }
258
259 if (llclose(fd_serial_port, RECEIVER) < 0) {
260     free(packet);
261     return -1;
262 }
263
264 free(packet);
265 return 0;
266 }

```

Listing 5: aplic.c

```

1 #include "linklayer.h"
2 #include <sys/types.h>
3 #include <sys/stat.h>
4 #include <fcntl.h>

```



```

5 #include <termios.h>
6 #include <stdio.h>
7 #include <string.h>
8 #include <strings.h>
9 #include <stdlib.h>
10 #include <unistd.h>
11 #include <signal.h>
12 #include <stdint.h>
13
14 #define FLAG 0x7E
15 #define ESC 0x7D
16 #define A 0x03
17 #define C_SET 0x03
18 #define C_DISC 0x0B
19 #define C_UA 0x07
20
21 #define C_RR(r) (0x05 | (((r) << 7) & 0x80))
22 #define C_REJ(r) (0x01 | (((r) << 7) & 0x80))
23 #define C_I(s) ((s) << 6)
24
25 #define CONTROL_SIZE 5
26
27 #define STUFFER 0x20
28
29 #define TIME_OUT_TIME 3
30 #define MAX_NO_TIMEOUT 3
31
32 #define HEADER_AND_TAIL_SIZE 10 // more than enough
33
34 typedef enum control_frame_type {
35     SET,
36     DISC,
37     UA,
38     RR,
39     REJ
40 } control_frame_type_t;
41
42 typedef enum state_sv_frame {
43     START,
44     FLAG_RCV,
45     A_RCV,
46     C_RCV,
47     RR_RCV,
48     REJ_RCV,
49     BCC_OK,
50     STOP
51 } state_sv_frame_t;
52
53 typedef enum state_info_rcv {
54     I_START,
55     I_GOT_FLAG,
56     I_IGNORE,
57     I_GOT_A,
58     I_GOT_C,

```

```

59     I_GOT_BCC1,
60     I_DATA_COLLECTION,
61     I_GOT_ESC,
62     I_GOT_END_FLAG,
63     I_TEST_DUP_RR,
64     I_TEST_DUP_REJ,
65     I_RR_DONT_STORE,
66     I_RR_STORE,
67     I_REJ,
68     I_STOP
69 } state_info_rcv_t;
70
71 static struct termios oldtio;
72 static volatile int g_count = 0;
73
74 static uint8_t S = 0;
75 static uint8_t next_S = 0;
76 static uint8_t R = 0;
77
78 static void control_frame_builder(control_frame_type_t cft,
79     uint8_t msg[]){
80     msg[0] = FLAG;
81     msg[1] = A;
82
83     switch (cft) {
84     case SET:
85         msg[2] = C_SET;
86         break;
87
88     case DISC:
89         msg[2] = C_DISC;
90         break;
91
92     case UA:
93         msg[2] = C_UA;
94         break;
95
96     case RR:
97         msg[2] = C_RR(R);
98         break;
99
100    case REJ:
101        msg[2] = C_REJ(R);
102        break;
103
104    default:
105        break;
106    }
107
108    msg[3] = msg[1] ^ msg[2];
109    msg[4] = FLAG;
110 }
111 static int update_state_rr_rej(state_sv_frame_t *state, uint8_t

```

```

byte) {
112
113     if (state == NULL) {
114         return 1;
115     }
116     switch (*state) {
117
118         case START:
119             if (byte == FLAG) *state = FLAG_RCV;
120             else *state = START;
121             break;
122
123         case FLAG_RCV:
124             if (byte == A) *state = A_RCV;
125             else *state = START;
126             break;
127
128         case A_RCV:
129             if ((byte & 0x0F) == 0x05) *state = RR_RCV;
130             else if ((byte & 0x0F) == 0x01) *state = REJ_RCV;
131             else *state = START;
132             next_S = (byte >> 7) & 0x01;
133             break;
134
135         case RR_RCV:
136             if (byte == (A^C_RR(next_S))) *state = BCC_OK;
137             else if (byte == FLAG) *state = FLAG_RCV;
138             else *state = START;
139             break;
140
141         case REJ_RCV:
142             if (byte == (A^C_REJ(next_S))) *state = BCC_OK;
143             else if (byte == FLAG) *state = FLAG_RCV;
144             else *state = START;
145             break;
146
147         case BCC_OK:
148             if (byte == FLAG) *state = STOP;
149             else *state = START;
150             break;
151
152         case STOP:
153             break;
154
155         default:
156             printf("ERROR: not supposed to reach this\n");
157             break;
158     }
159
160     return 0;
161 }
162
163 static int update_state_set_ua(uint8_t c, state_sv_frame_t *
    state, uint8_t byte) {

```

```

164     if (state == NULL) {
165         return 1;
166     }
167
168     switch (*state) {
169         case START:
170             if (byte == FLAG) {
171                 *state = FLAG_RCV;
172             }
173             break;
174
175         case FLAG_RCV:
176             if (byte == A) {
177                 *state = A_RCV;
178             } else if (byte != FLAG) {
179                 *state = START;
180             }
181             break;
182
183         case A_RCV:
184             if (byte == c) {
185                 *state = C_RCV;
186             } else if (byte == FLAG) {
187                 *state = FLAG_RCV;
188             } else {
189                 *state = START;
190             }
191             break;
192
193         case C_RCV:
194             if (byte == (A^c)) {
195                 *state = BCC_OK;
196             } else if (byte == FLAG) {
197                 *state = FLAG_RCV;
198             } else {
199                 *state = START;
200             }
201             break;
202
203         case BCC_OK:
204             if (byte == FLAG) {
205                 *state = STOP;
206             } else {
207                 *state = START;
208             }
209             break;
210
211         case STOP:
212             break;
213
214         default:
215             printf("ERROR: not supposed to reach this\n");
216             break;
217     }

```

```

218
219
220     return 0;
221 }
222
223 static int update_state_info_rcv(state_info_rcv_t *state,
    uint8_t byte){
224
225     switch (*state){
226     case (I_START):
227         if (byte == FLAG) *state = I_GOT_FLAG;
228         else *state = I_IGNORE;
229         break;
230
231     case (I_GOT_FLAG):
232         if (byte == FLAG) *state = I_GOT_FLAG;
233         else if (byte == A) *state = I_GOT_A;
234         else *state = I_IGNORE;
235         break;
236
237     case (I_IGNORE):
238         if (byte == FLAG) *state = I_GOT_FLAG;
239         else *state = I_IGNORE;
240         break;
241
242     case (I_GOT_A):
243         if (byte == C_I(R)) *state = I_GOT_C;
244         else *state = I_IGNORE;
245         break;
246
247     case (I_GOT_C):
248         if (byte == (C_I(R)^A)) *state = I_GOT_BCC1;
249         else *state = I_IGNORE;
250         break;
251
252     case (I_GOT_BCC1):
253         if (byte == ESC) *state = I_GOT_ESC;
254         else *state = I_DATA_COLLECTION;
255         break;
256
257     case (I_DATA_COLLECTION):
258         if (byte == FLAG) *state = I_GOT_END_FLAG;
259         else if (byte == ESC) *state = I_GOT_ESC;
260         else *state = I_DATA_COLLECTION;
261         break;
262
263     case (I_GOT_ESC):
264         *state = I_DATA_COLLECTION;
265         break;
266
267     case (I_GOT_END_FLAG):
268         if (byte) *state = I_TEST_DUP_RR; // byte = is bcc2
    valid ?
269         else *state = I_TEST_DUP_REJ;

```

```

270         break;
271
272         case (I_TEST_DUP_RR):
273             if (byte) *state = I_RR_DONT_STORE; // byte = is
dup ?
274             else *state = I_RR_STORE;
275             break;
276
277         case (I_TEST_DUP_REJ):
278             if (byte) *state = I_RR_DONT_STORE; // byte = is
dup ?
279             else *state = I_REJ;
280             break;
281
282         case (I_RR_DONT_STORE):
283             *state = I_START;
284             break;
285
286         case (I_RR_STORE):
287             *state = I_STOP;
288             break;
289
290         case (I_REJ):
291             *state = I_START;
292             break;
293
294         case (I_STOP):
295             break;
296
297         default:
298             break;
299     }
300
301     return 0;
302 }
303
304 static void time_out() {
305     printf("alarm # %d\n", g_count);
306     g_count++;
307 }
308
309 static int setup_alarm() {
310     struct sigaction new;
311     sigset_t smask;
312
313     if (sigemptyset(&smask)==-1) {
314         perror ("sigsetfunctions");
315         return 1;
316     }
317
318     new.sa_handler = time_out;
319     new.sa_mask = smask;
320     new.sa_flags = 0;
321

```

```

322     if (sigaction(SIGALRM, &new, NULL) == -1) {
323         perror ("sigaction");
324         return 1;
325     }
326
327     return 0;
328 }
329
330 static int common_open(int porta) {
331     int fd = -1;
332     struct termios newtio;
333
334     /*
335     Open serial port device for reading and writing and not as
336     controlling tty
337     because we don't want to get killed if linenoise sends CTRL
338     -C.
339     */
340
341     char buffer[20];
342     if (sprintf(buffer, "/dev/ttyS%d", porta) < 0) {
343         perror("");
344         return -1;
345     }
346
347     fd = open(buffer, O_RDWR | O_NOCTTY );
348     if (fd < 0) {
349         perror(buffer);
350         return -1;
351     }
352
353     if (tcgetattr(fd, &oldtio) == -1) { /* save current port
354     settings */
355         perror("tcgetattr");
356         return -1;
357     }
358
359     bzero(&newtio, sizeof(newtio));
360     newtio.c_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;
361     newtio.c_iflag = IGNPAR;
362     newtio.c_oflag = 0;
363
364     /* set input mode (non-canonical, no echo,...) */
365     newtio.c_lflag = 0;
366
367     newtio.c_cc[VTIME]      = 0;   /* inter-character timer
368     unused */
369     newtio.c_cc[VMIN]       = 1;   /* blocking read until 1 char
370     received */
371
372     tcflush(fd, TCIOFLUSH);
373
374     if (tcsetattr(fd,TCSANOW,&newtio) == -1) {

```

```

371         perror("tcsetattr");
372         return -1;
373     }
374
375     printf("New termios structure set\n");
376
377     return fd;
378 }
379
380 static int common_close(int fd) {
381     if (tcsetattr(fd, TCSANOW, &oldtio) == -1) {
382         perror("tcsetattr");
383         close(fd);
384         return -1;
385     }
386
387     return close(fd);
388 }
389
390 static void R_invert(){
391     R = ((!R) << 7) >> 7;
392 }
393
394 int llopen(int porta, type_t type) {
395     state_sv_frame_t state;
396     int fd = common_open(porta);
397     if (fd < 0) {
398         printf("Failed to open serial port.\n");
399         return -1;
400     }
401
402     printf("%d opened fd\n", fd);
403
404     uint8_t set[CONTROL_SIZE];
405     control_frame_builder(SET, set);
406
407     uint8_t ua[CONTROL_SIZE];
408     control_frame_builder(UA, ua);
409
410     if (setup_alarm() != 0) {
411         common_close(fd);
412         return -1;
413     }
414     g_count = 0;
415
416     switch (type) {
417     case TRANSMITTER:;
418         int ua_received = FALSE;
419         int res;
420         while (g_count < MAX_NO_TIMEOUT && !ua_received) {
421             state = START;
422
423             res = write(fd, set, CONTROL_SIZE * sizeof(uint8_t)
);

```



```

424         if (res == -1) {
425             printf("llopen() -> write() TRANSMITTER error\n
");
426             common_close(fd);
427             return -1;
428         }
429         printf("SET sent.\n");
430         printf("%d bytes written\n", res);
431
432         alarm(TIME_OUT_TIME);
433
434         int timed_out = FALSE;
435         while (!timed_out && state != STOP) {
436             uint8_t byte_read = 0;
437
438             res = read(fd, &byte_read, 1);
439             if (res == 1) {
440                 if (update_state_set_ua(C_UA, &state,
byte_read) != 0) {
441                     common_close(fd);
442                     alarm(0);
443                     return -1;
444                 }
445                 ua_received = (state==STOP);
446
447                 } else if (res == -1) {
448                     timed_out = TRUE;
449
450                 } else {
451                     printf("DEBUG: not supposed to happen\n");
452                 }
453             }
454
455             alarm(0);
456         }
457
458         if (ua_received) {
459             printf("UA received.\n");
460             printf("ACK\n");
461         } else {
462             common_close(fd);
463             return -1;
464         }
465
466         break;
467
468     case RECEIVER:
469         alarm(TIME_OUT_TIME * MAX_NO_TIMEOUT);
470         state = START;
471         while (state != STOP) {
472             uint8_t byte_read = 0;
473             res = read(fd, &byte_read, 1);
474
475             if (res == 1) {

```

```

476         if (update_state_set_ua(C_SET, &state,
byte_read) != 0) {
477             common_close(fd);
478             alarm(0);
479             return -1;
480         }
481     } else if (res == -1) {
482         if (g_count > 0) {
483             printf("llopen timeout\n");
484         } else {
485             printf("llopen() -> read() RECEIVER error\n
");
486         }
487         common_close(fd);
488         alarm(0);
489         return -1;
490     } else {
491         printf("DEBUG: not supposed to happen\n");
492     }
493 }
494
495 printf("SET received.\n");
496 if (write(fd, ua, CONTROL_SIZE) < 0) {
497     printf("llopen() -> write() RECEIVER error\n");
498     common_close(fd);
499     alarm(0);
500     return -1;
501 }
502
503 printf("UA sent.\n");
504 printf("ACK\n");
505 alarm(0);
506 break;
507 }
508
509 return fd;
510 }
511
512 int llclose(int fd, type_t type) {
513     state_sv_frame_t state;
514     uint8_t disc[CONTROL_SIZE];
515     control_frame_builder(DISC, disc);
516
517     uint8_t ua[CONTROL_SIZE];
518     control_frame_builder(UA, ua);
519
520     int res = 0;
521     int disc_received = FALSE;
522     g_count = 0;
523
524     switch (type) {
525     case TRANSMITTER:
526         while (g_count < MAX_NO_TIMEOUT && !disc_received) {
527             state = START;

```

```

528
529         res = write(fd, disc, CONTROL_SIZE * sizeof(uint8_t
530     ));
531     if (res == -1) {
532         printf("llclose() -> write() TRANSMITTER error\
533     n");
534         return -1;
535     }
536     printf("DISC sent.\n");
537     printf("%d bytes written\n", res);
538
539     alarm(TIME_OUT_TIME);
540
541     int timed_out = FALSE;
542     while (!timed_out && state != STOP) {
543         uint8_t byte_read = 0;
544
545         res = read(fd, &byte_read, 1);
546         if (res == 1) {
547             if (update_state_set_ua(C_DISC, &state,
548         byte_read) != 0) {
549                 common_close(fd);
550                 alarm(0);
551                 return -1;
552             }
553             disc_received = (state==STOP);
554
555             } else if (res == -1) {
556                 timed_out = TRUE;
557
558             } else {
559                 printf("DEBUG: not supposed to happen\n");
560             }
561         }
562
563         alarm(0);
564
565         if (disc_received) {
566             printf("DISC received.\n");
567             res = write(fd, ua, CONTROL_SIZE * sizeof(uint8_t))
568         ;
569             printf("UA sent.\n");
570         } else {
571             printf("DISC not received.\n");
572         }
573
574         break;
575
576     case RECEIVER:
577         alarm(TIME_OUT_TIME * MAX_NO_TIMEOUT);
578         state = START;
579         while (state != STOP) {
580             uint8_t byte_read = 0;

```

```

578         res = read(fd, &byte_read, 1);
579         if (res == 1) {
580             if (update_state_set_ua(C_DISC, &state,
581 byte_read) != 0) {
582                 common_close(fd);
583                 alarm(0);
584                 return -1;
585             }
586             disc_received = (state==STOP);
587         } else if (res == -1) {
588             if (g_count > 0) {
589                 printf("llclose timedout\n");
590             } else {
591                 printf("llclose() -> read() RECEIVER error\
n");
592             }
593             alarm(0);
594             common_close(fd);
595             return -1;
596             break;
597         } else {
598             printf("DEBUG: not supposed to happen\n");
599         }
600     }
601
602     alarm(0);
603
604     if (res != -1) {
605         printf("DISC received.\n");
606         int ua_received = FALSE;
607         g_count = 0;
608         while (g_count < MAX_NO_TIMEOUT && !ua_received) {
609             state = START;
610
611             res = write(fd, disc, CONTROL_SIZE * sizeof(
uint8_t));
612             if (res == -1) {
613                 printf("llclose() -> write() RECEIVER error
\n");
614             }
615             alarm(0);
616             return -1;
617         }
618         printf("DISC sent.\n");
619         printf("%d bytes written\n", res);
620
621         alarm(TIME_OUT_TIME);
622
623         int timed_out = FALSE;
624         while (!timed_out && state != STOP) {
625             uint8_t byte_read = 0;
626
627             res = read(fd, &byte_read, 1);
628             if (res == 1) {

```

```

628         if (update_state_set_ua(C-UA, &state,
byte_read) != 0) {
629             common_close(fd);
630             alarm(0);
631             return -1;
632         }
633         ua_received = (state==STOP);
634
635         } else if (res == -1) {
636             timed_out = TRUE;
637
638         } else {
639             printf("DEBUG: not supposed to happen\n
");
640         }
641     }
642
643     alarm(0);
644 }
645
646     if (ua_received) {
647         printf("UA received.\n");
648     }
649 }
650
651     break;
652 }
653
654     res = common_close(fd);
655
656     return res;
657 }
658
659 int message_stuffing(uint8_t in_msg[], unsigned int in_msg_size
, uint8_t ** out_msg){
660
661     int size_counter = 0;
662     *out_msg = malloc(in_msg_size*2);
663
664     uint8_t * out_message = * out_msg;
665
666     for (int i = 0; i < in_msg_size; i++){
667         switch (in_msg[i]){
668             case FLAG:
669                 out_message[size_counter++] = ESC;
670                 out_message[size_counter++] = FLAG ^ STUFFER;
671                 break;
672             case ESC:
673                 out_message[size_counter++] = ESC;
674                 out_message[size_counter++] = ESC ^ STUFFER;
675                 break;
676             default:
677                 out_message[size_counter++] = in_msg[i];
678                 break;

```

```

679     }
680 }
681 return size_counter;
682 }
683
684 int message_destuffer(uint8_t in_msg[], unsigned int
in_msg_size, uint8_t ** out_msg){
685
686     int size_counter = 0;
687     *out_msg = malloc(in_msg_size);
688
689     uint8_t * out_message = * out_msg;
690
691     for (int i = 0; i < in_msg_size; i++){
692         if (in_msg[i] == ESC){
693             out_message[size_counter] = (in_msg[++i] ^ STUFFER)
;
694         } else {
695             out_message[size_counter] = in_msg[i];
696         }
697         size_counter++;
698     }
699
700     return size_counter;
701 }
702
703 uint8_t bcc2_builder(uint8_t msg[], unsigned int msg_size){
704
705     if (msg_size == 1) {
706         return msg[0];
707     } else if ( msg_size < 0) {
708         return 0;
709     }
710
711     uint8_t ret = msg[0];
712
713     for (int i = 1; i < msg_size; i++){
714         ret ^= msg[i];
715     }
716
717     return ret;
718 }
719
720 int llwrite(int fd, uint8_t * buffer, int length){
721
722     int write_successful = 0;
723     int ret = 0;
724     uint8_t bcc2 = bcc2_builder(buffer, length);
725     uint8_t *unstuffed_msg = malloc((length+1) * sizeof(uint8_t
));
726     memcpy(unstuffed_msg, buffer, length);
727     unstuffed_msg[length] = bcc2;
728     uint8_t *stuffed_msg = NULL;
729     int stuffed_msg_len = message_stuffing(unstuffed_msg,

```

```

length+1, &stuffed_msg);
730 free(unstuffed_msg);
731 int total_msg_len = stuffed_msg_len + CONTROL_SIZE;
732 uint8_t *info_msg = malloc(total_msg_len);
733
734 info_msg[0] = FLAG;
735 info_msg[1] = A;
736 info_msg[2] = C_I(S);
737 info_msg[3] = A ^ C_I(S);
738 memcpy(&(info_msg[4]), stuffed_msg, stuffed_msg_len);
739 info_msg[total_msg_len-1] = FLAG;
740
741 setup_alarm();
742 g_count = 0;
743
744 while(!write_successful && g_count < MAX_NO_TIMEOUT) {
745
746     printf("----- TASK: WRITING MESSAGE\n");
747
748     if (write(fd, info_msg, total_msg_len * sizeof(uint8_t)
749 ) == -1) {
750         printf("llwrite() -> write() error\n");
751         free(info_msg);
752         free(stuffed_msg);
753         return -1;
754     }
755
756     printf("----- TASK: DONE\n");
757
758     uint8_t byte_read = 0;
759     int res = 0;
760     state_sv_frame_t state = START;
761
762     printf("----- TASK: READING REPLY\n");
763
764     alarm(TIME_OUT_TIME);
765
766     while(state != STOP){
767         res = read(fd, &byte_read, 1);
768
769         if (res == -1) {
770             write_successful = 0;
771             break;
772         }
773
774         update_state_rr_rej(&state, byte_read);
775         printf("BYTE: 0x%x; STATE: %d\n", byte_read, state)
776
777         ;
778
779         if (state == RR_RCV) {
780             write_successful = 1;
781         } else if (state == REJ_RCV) {
782             write_successful = 0;
783         }
784     }

```

```

781     }
782
783     printf("----- TASK: DONE\n");
784 }
785
786 alarm(0);
787
788 S = next_S;
789
790 free(info_msg);
791 free(stuffed_msg);
792
793 if (g_count >= MAX_NO_TIMEOUT) {
794     ret = -1;
795 } else {
796     ret = total_msg_len;
797 }
798
799 return ret;
800 }
801
802 int llread(int fd, uint8_t *buffer) {
803
804     state_info_rcv_t state;
805     uint8_t byte_read = 0;
806     uint8_t data_read[DATA_PACKET_MAX_SIZE * 2 +
807     HEADER_AND_TAIL_SIZE];
808     int msg_size = 0;
809
810     uint8_t *unstuffed_msg = NULL;
811     int unstuffed_size = 0;
812
813     setup_alarm();
814     g_count = 0;
815
816     state = I_START;
817
818     printf("--- NEW READ ---\n");
819
820     while (state != I_STOP){
821
822         printf("--- TRY READ ---\n");
823
824         alarm(TIME_OUT_TIME * MAX_NO_TIMEOUT);
825
826         msg_size = 0;
827         int rcv_s = -1;
828
829         while (state != I_GOT_BCC1){
830             printf("PHASE 1 ; START_STATE : %d ; ", state);
831             if (read(fd, &byte_read, 1) == -1) {
832                 printf("llread() -> read() 1. error.\n");
833                 alarm(0);
834                 free(unstuffed_msg);

```



```

834         return -1;
835     }
836
837     if (g_count) {
838         alarm(0);
839         free(unstuffed_msg);
840         return -1;
841     }
842     if (state == I_GOT_C) rcv_s = byte_read >> 6;
843     update_state_info_rcv(&state, byte_read);
844     printf("END_STATE : %d\n", state);
845 }
846
847 while(state != I_GOT_END_FLAG) {
848     printf("PHASE 2 ; START_STATE : %d ; ", state);
849     if (read(fd, &byte_read, 1) == -1) {
850         printf("llread() -> read() 2. error.\n");
851         alarm(0);
852         free(unstuffed_msg);
853         return -1;
854     }
855
856     if (g_count) {
857         alarm(0);
858         free(unstuffed_msg);
859         return -1;
860     }
861     update_state_info_rcv(&state, byte_read);
862     data_read[msg_size] = byte_read;
863     msg_size++;
864     printf("BYTE : 0x%x ; END_STATE : %d\n", byte_read,
state);
865 }
866
867     unstuffed_size = 0;
868     uint8_t rej_msg[CONTROL_SIZE];
869     uint8_t rr_msg[CONTROL_SIZE];
870
871     free(unstuffed_msg);
872     unstuffed_size = message_destuffer(data_read, msg_size
-1, &unstuffed_msg);
873
874     while (state != I_STOP && state != I_START){
875
876         printf("PHASE 3 ; START_STATE : %d ; ", state);
877
878         uint8_t res = 0;
879
880         switch(state){
881             case (I_GOT_END_FLAG):
882                 res = unstuffed_msg[unstuffed_size-1] ==
bcc2_builder(unstuffed_msg, unstuffed_size-1);
883                 break;
884

```

```

885         case (I_TEST_DUP_REJ):
886             res = rcv_s != R;
887             break;
888
889         case (I_TEST_DUP_RR):
890             res = rcv_s != R;
891             break;
892
893         case (I_RR_DONT_STORE):
894             R_invert();
895             control_frame_builder(RR, rr_msg);
896             if (write(fd, rr_msg, CONTROL_SIZE) == -1)
897             {
898                 printf("llread() -> write() 1. error.\n");
899
900                 alarm(0);
901                 free(unstuffed_msg);
902                 return -1;
903             }
904             break;
905
906         case (I_RR_STORE):
907             R_invert();
908             control_frame_builder(RR, rr_msg);
909             memcpy(buffer, unstuffed_msg,
910 unstuffed_size-1);
911             if (write(fd, rr_msg, CONTROL_SIZE) == -1)
912             {
913                 printf("llread() -> write() 2. error.\n");
914
915                 alarm(0);
916                 free(unstuffed_msg);
917                 return -1;
918             }
919             break;
920
921         case (I_REJ):
922             control_frame_builder(REJ, rej_msg);
923             if (write(fd, rej_msg, CONTROL_SIZE) == -1)
924             {
925                 printf("llread() -> write() 3. error.\n");
926
927                 alarm(0);
928                 free(unstuffed_msg);
929                 return -1;
930             }
931             break;
932
933         case (I_STOP):
934             break;
935
936         case (I_START):
937             break;

```

```

932         default:
933             printf("NOT SUPPOSED TO REACH THIS\n");
934             break;
935     }
936     update_state_info_rcv(&state, res);
937     printf("RES : %d ; END_STATE : %d\n", res, state);
938 }
939 }
940
941 alarm(0);
942
943 free(unstuffed_msg);
944
945 return msg_size;
946 }

```

Listing 6: linklayer.c