Protocolo Ligação de Dados

1º Trabalho Laboratorial

Relatório Final

Mestrado Integrado em Engenharia Informática e Computação

Redes de Computadores

Turma - 7

Hugo Vaz Neves - up201104178@fe.up.pt

Nuno Manuel Ferreira Corte-Real – up201405158@fe.up.pt

Pedro Azevedo - up201306026@fe.up.pt

Faculdade de Emgemharia da Universidade do Porto

Rua Roberto Frias, s/n, 4200-465 Porto, Portugal

13 de Novembro de 2017

Índice:

1 Sumário

Este relatório serve como guia para o primeiro projeto laboratorial da unidade curricular de Redes de Computadores do curso MIEIC da FEUP. O objetivo deste projeto era implementar um protocolo de ligação de dados sobre a porta de série RS-232, evitando que informação se perca ou corrompa quando se introduzem interferências físicas durante a transmissão de informação.

O código realizado é capaz de transmitir qualquer ficheiro (.png, .gif, .jpg, .txt, etc.) sem erros. Aquando interrupção da transmissão de dados na porta de série, o pacote interrompido é ignorado e o programa re-envia os dados, transmitindo o ficheiro com sucesso. Aquando interferências físicas por curto-circuito, o programa efetua um processo semelhante, transmitindo também o ficheiro com sucesso. Quando há uma interrupção demasiado demorada, o programa dá *time-out* e termina.

2 Introdução

O principal objetivo deste trabalho era implementar um protocolo de ligação de dados em C, usando o ambiente Linux. O envio do ficheiro deveria ser realizado por meio de tramas de informação, portadoras da informação do ficheiro e tramas de supervisão, usadas para assegurar que a informação é enviada sem erros. Para a conexão entre transmissor e receptor, foi utilizada uma ligação assíncrona, por meio de uma porta de série RS-232, configurada em modo não canónico. O guião do trabalho requeria a implementação de 4 funções principais para a instalação do protocolo: llopen, llread, llwrite e llclose.

O relatório será dividido nas seguintes secções:

- Introdução
- Arquitetura
- Estrutura do Código
- Casos de Uso Principais
- Protocolo de Ligação Lógica
- Protocolo de Aplicação
- Validação
- Eficiência do protocolo de ligação de dados
- Conclusão

3 Arquitetura

O código foi dividido em duas camadas principais. A camada de ligação, alusiva à transmissão e recepção de dados, implementada nos ficheiros *link_layer.c* e *link_layer.h* e a camada da aplicação, alusiva a funções de manipulação de ficheiros, servindo de intermédio entre os dados e a camada de ligação.

Deste modo, a camada de ligação possui em si as 4 funções principais do projeto, que estabelecem e fecham a ligação (llopen e llclose, respetivamente) e que enviam e recebem dados (llwrite e llread, respetivamente). Possui também funções auxiliares para a implementação da verificação de dados (*stuffing*, *destuffing*,*calculate*BCC2 e *switch*C1).

A camada da aplicação reúne em si todas as funções que tratam da abertura, manipulação e obtenção de informação do ficheiro a transferir/ler (*open_file*, *file_size*, *check_num_bytes*, *ge_file_info*, *create_file*, *get_data*,

create_start_end_package, *create_data_package*) e que servem de intermédio entre o utilizador e a camada de ligação (*transmitter e receiver*).

Para transmitir um ficheiro, é preciso executar o programa duas vezes, uma em modo *TRANSMITTER*e outra em modo *RECEIVER*, em dois computadores diferentes ligados por uma porta de série ou em duas máquinas virtuais em portas de série virtuais diferentes.

4 Estrutura do Código

4.1 Camada de Ligação de Dados

Nos ficheiros *link_layer.c* e *link_layer.h* está implementada a camada de ligação de dados. As principais funções da camada de ligação são as seguintes:

```
int llopen(int port, char mode);
int llwrite(int fd, char *buffer, int len);
int llread(int fd, char *buffer);
int llclose(int fd, int flag);
```

llopen estabelece a ligação, enviando uma mensagem SET e recebendo uma mensagem UA (tramas de supervisão) caso seja invocada pelo Transmissor e recebendo uma mensagem SET e enviando uma mensagem UA caso seja invocada pelo Receptor.

Llwrite envia tramas de informação que carregam os dados do ficheiro a enviar. Antes de enviar a informação processa-a através de um mecanismo de *stuffing*.

Llread lê uma trama de informação, processa-a pelo mecanismo de *destuffing* e verifica se foi enviada sem erros.

Foram implementadas ainda as seguintes funções auxiliares:

```
int getRR();
void setRR();
/* Utilities */
int stuffing(char * package, int length);
int deStuffing(char * package, int length);
void switchC1();
 char calculateBCC2(char* buffer,  int size);
```

stuffing realiza o mecanismo do mesmo nome

destuffing desfaz o mecanismo de *stuffing*

*calculate*BCC2 realiza o ou exclusivo de todos os elementos de um char*

*switch*C1 muda o número sequencial atual

*get*RR retorna a uma flag que indica se uma trama de informação foi rejeitada

*set*RR reinicia o valor da flag acima citada

4.2 Camada da Aplicação

Nos ficheiros *application_layer.c* e *application_layer.h* está implementada a camada da aplicação. As funções da camada da aplicação são as seguintes:

```
int transmitter(char * fileName, int fd);
int receiver(int fd);
int open_file(FILE ** file, char * fileName);
unsigned long file_size(FILE * file,  int * fileSize);
int create_start_end_package(int type, char * fileName,  int size, char * package);
int check_num_bytes(int size);
int get_file_info(char* buffer, int buffsize,  int *size, char *name);
int create_file(FILE ** file, char * fileName);
int get_data (char * buffer, int size);
int create_data_package(char *buffer, int size, char packageID);
```

transmitter - abre e processa um ficheiro e envia-o para o recetor através do llwrite

receiver - recebe um ficheiro do transissor e processa-o, reenviando um pacote caso este possua erros

Fora implementadas as seguintes funções auxiliares:

open_file - Função encarregada de abrir um ficheiro, caso este exista

file_size - Verifica o tamanho efectivo de um ficheiro

create_start_end_package - cria as tramas de supervisão

check_num_bytes - conta o número de bytes que o ficheiro a transmitir tem

get_file_info - processa um ficheiro e extrai dele informação sobre ele, nomeadamente o número de bytes, nome e dados a enviar

create_file - cria o ficheiro consoante os dados recebidos do ficheiro a ser enviado

get_data - extrai os dados do ficheiro da trama de informação

create_data_package - cria a trama de informação a mandar a partir dos dados do ficheiro

5 Protocolo de Ligação Lógica

No início do programa é executada a função *llopen*, dividida em parte do transmissor e parte do receptor. O transmissor envia uma mensagem SET ao receptor que a processa via uma máquina de estados e recebe uma mensagem de confirmação UA, processando-a via um máquina e estados semelhante. Caso este processo seja realizado com sucesso, a conexão fica estabelecida.

De seguida é invocada a função *llwrite*, que recebe como argumento o descritor de ficheiro retornado pelo *llopen*, um buffer e o tamanho do buffer. A primeira tarefa da função llwrite é invocar a função *calculate*BCC2 para realizar o XOR de todos os elementos do buffer, de modo a obter o BCC2 para a trama de informação que se vai enviar.

De seguida é realizado o *stuffing* do *buffer* enviado como argumento, através da função *stuffing*, que retorna o novo tamanho do *buffer* pós-*stuffing*. O próximo passo é "montar" a trama de informação a enviar, e efectivamente enviá-la.

A seguir é activado o alarme e aguarda-se a resposta do *llread*, que vai confirmar se a informação foi enviada sem erros.

Segue-se a invocação do *llread*. Esta função entra num *loop* em que espera por *bytes* vindos do *llwrite* e processa-os numa máquina de estados. Esta é reiniciada sempre que processa um *byte* não esperado. Na máquina são guardados os dados relativos à informação do ficheiro, procedendo-se ao *destuffing* da informação armazenada através da função *deStuffing*. O BCC2 é recalculado com a informação pós-*destuffing* e compara-a com o BCC2 recebido originalmente, escrevendo para o *llwrite* uma mensagem de confirmação (RR) ou de rejeição (REJ).

Por fim, é invocada a função *llclose*. Nesta função, é enviado o comando Disc pelo recetor que indica o fim da ligação e fica à espera de uma resposta com um comando DISC igual. Ao receber o DISC, este é processado numa máquina de estados adaptada para tal e em caso de sucesso, é enviada uma mensagem UA, processada por outra máquina de estados. Em caso de sucesso, o programa termina.

É de salientar que foram criadas duas funções *handler* de sinais, uma para o transmissor (*handle*) e uma para o receptor (*receiverHandle*), acionadas, respectivamente, no *llwrite* e no *llread*. Cada uma destas funções faz a contagem das vezes que o programa tenta reenviar informação torna a enviar essa informação.

6 Protocolo da Aplicação

A camada de aplicação serve como intermédio entre o utilziador, a manipulação de ficheiros e a camada da ligação de dados. Esta camada está dividida em duas partes, cada uma com a sua função principal, a do transmissor e a do receptor.

A parte do transmissor é representada pela função *transmitter*, cujos argumentos são o nome do ficheiro a transmitir e o descritor do ficheiro retornado pelo *llopen*. Nesta função, é aberto o ficheiro (através da função *open_file*) e de seguida utiliza-se a função *file_size* para se obter o tamanho do ficheiro.

Segue-se o fabrico da trama da informação através da função *create_start_end_package*, que recebe como argumentos uma flag a indicar se a trama é de *START* ou *END*, o nome do ficheiro, o seu tamanho. Criada a trama, esta é enviada para o *receiver* através do *llwrite*.

Seguidamente, é implementado o ciclo que vai ler o ficheiro, criar tramas de informação e re-enviá-las, em caso de erro. O ciclo repete-se, no máximo, um número de vezes igual ao número de tentativas de re-envio configuradas. Por fim, retorna.

A parte do receptor é representada pela função *receiver*, que recebe como argumento o descritor do ficheiro retornado pelo *llopen*. Esta começa por invocar a função *get_file_info*, que extrai do *start package* informação relativa ao ficheiro. De seguida, invoca a função *create_file*, que tenta abrir o ficheiro segundo as informações recebidas. A seguir, entra num ciclo no qual invoca *llread* de modo a ler tramas de informação até ter lido um número de *bytes* igual ao número de *bytes* do ficheiro. Por cada trama, invoca a função *get_data* para extrair os dados do ficheiro da trama de informação e verifica a mensagem de confirmação por intermédio de uma *flag*, através da função *getRR* (RR ou REJ) e altera a *flag* conforme a mensagem recebida, através da função *setRR*. Para finalizar, lê o *end package* e retorna.

7 Casos de Uso Principais

A nossa aplicação pode ser utilizada e modo receptor e em modo emissor, ambos os modos trabalhando em união, tendo definido o nosso baudrate no header da link layer, e o tamanho dos pacotes no header na application layer.

Em modo de receptor é necessário dar início na consola ao modo de recepção. Em modo emissor o nome do ficheiro é introduzido na linha de comandos como input para o programa.

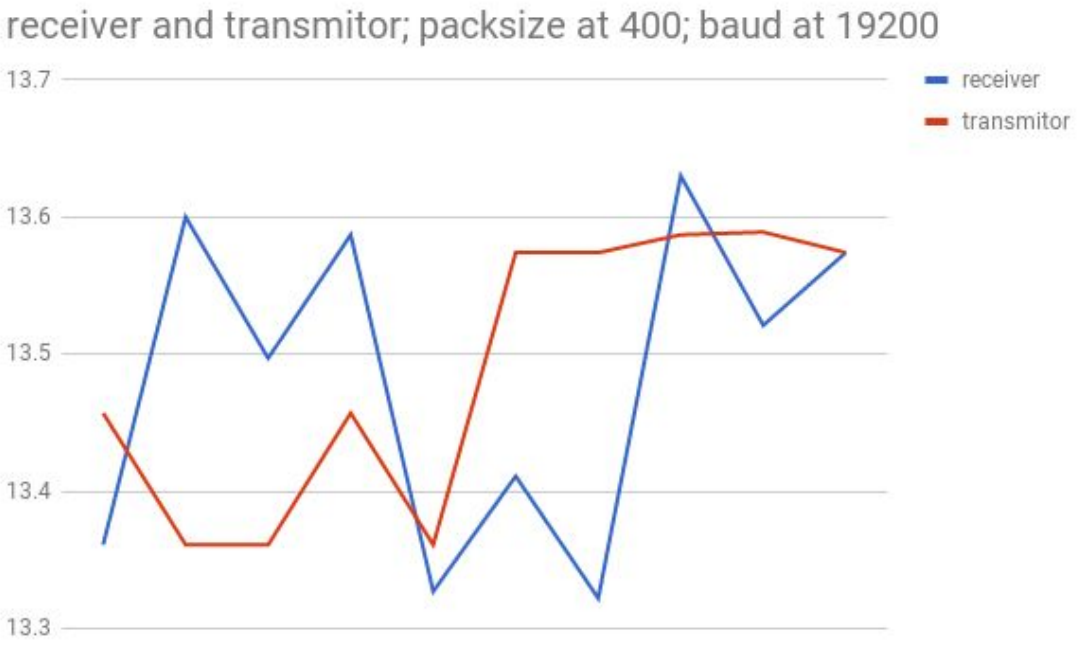
8 Validação

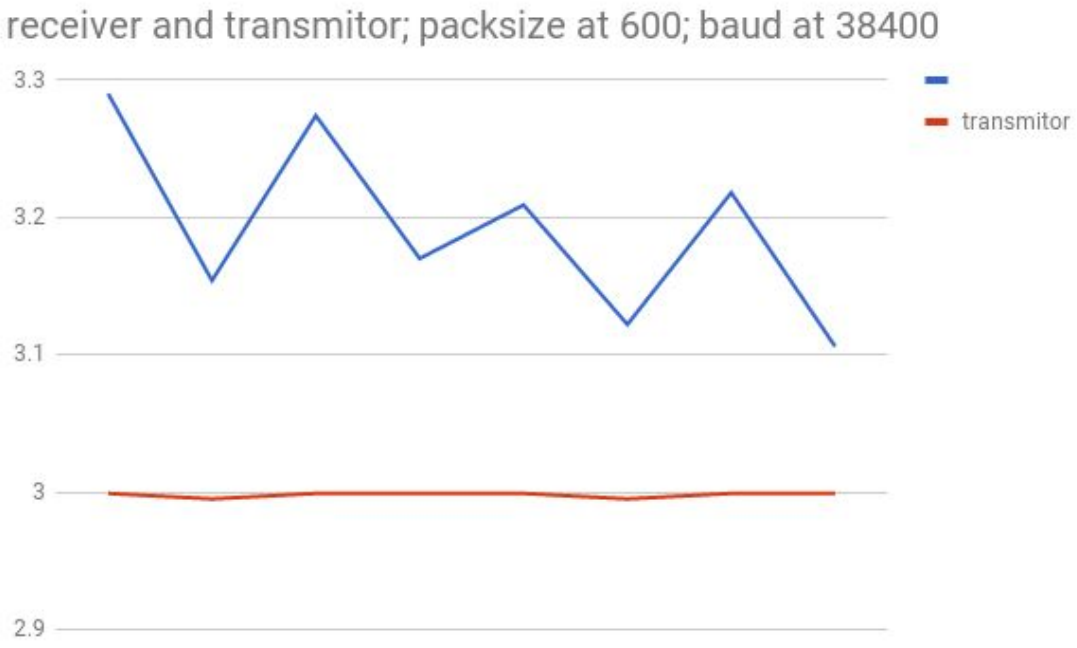
A validação do nosso projecto foi efectuada pela transmissão de ficheiros entre os dois computadores sob diferentes condições. O primeiro ficheiro a ser testado foi o "*pinguim.gif*" fornecido pelo enunciado. de seguida foram usados um ficheiro simples de texto, e um ficheiro *jpeg*; *Ambos enviados com sucesso.*

Seguimos com testes em que, usando o material disponível, criamos disrupções a nível de hardware. Estes testes consistiram da criação de mau contacto usando um fio de cobre e o uso de um switch button para cortar completamente a ligação.

O nosso programa passou em todos os testes. Após a transferência dos ficheiros em casos de testes onde foram registadas disrupções, tanto a qualidade dos ficheiros como o seu tamanho foi verificado como sendo o mesmo antes e após a sua transferência.

9 Eficiência do Protocolo de Ligação de Dados





packsize at 600	baud at 38400
receiver	transmitor
3.29	2.999
3.154	2.995
3.274	2.999

3.17	2.999
3.209	2.999
3.122	2.995
3.218	2.999
3.106	2.999
3.226	2.998
3.106	2.998

packsize at 400	baud at 19200
receiver	transmitor
13.361	13.457
13.6	13.361
13.497	13.361
13.587	13.457
13.327	13.361
13.411	13.574
13.322	13.574
13.63	13.587
13.521	13.589
13.574	13.574

10 Conclusões

Sentimos que o projecto foi um bom teste para a aplicação dos nossos conhecimentos da cadeira. A melhor decisão que tomamos foi no início do trabalho estruturar a nossa visão para como desenvolver o projecto. Contudo, com o desenvolvimento do nosso codigo, foi mais difícil manter a coesão.

No final, o nosso projecto passa com sucesso nos critérios pedidos pelo enunciado, e sentimos que absorvemos os conhecimentos da primeira parte dos conhecimentos da cadeira.

11 Anexos

link_layer.h

```
#ifndef LINK_LAYER_H
#define LINK_LAYER_H

typedef enum {TRANSMITTER, RECEIVER} UserMode;

int llopen(int port, char mode);
int llwrite(int fd, char *buffer, int len);
int llread(int fd, char *buffer);
int llclose(int fd, int flag);

int getRR();
void setRR();
/* Utilities */
int stuffing(char * package, int length);
int deStuffing(char * package, int length);
void switchC1();
 char calculateBCC2(char* buffer,  int size);

#define BAUDRATE B38400
#define _POSIX_SOURCE 1 /* POSIX compliant source */
#define FALSE 0
#define TRUE  1

#define TRANSMITTER 0
#define RECEIVER    1

#define RETRY_NUM   4

#define FLAG  0x7e
#define A     0x03
#define C_SET 0x03
#define C_DISC 0x09
#define UA    0x07

/* SET State Machine */
#define SET_SEND 0
#define START    1
#define FLAG_RCV 2
#define A_RCV    3
#define C_RCV    4
#define BCC_OK   5
#define END        6

/* UA State Machine */
#define UA_RCV   7

/* llread State Machine */
#define C1_RCV           8
#define BCC1_OK          9
#define BCC2_OK          10
#define DATA_PROCESSING  11

/* I frame */
#define ESC         0x7D
#define SUB         0x20
#define XOR_7E_20 0x5E
#define XOR_7D_20 0x5D
/*
#define RR        0x05
#define REJ       0x01
*/

#define MAXRETRIES 3

#endif
```

link_layer.c

```
63
64  int getRR()
65  {
66      return rrNotSend;
67  }
68  void setRR()
69  {
70      rrNotSend = 0;
71  }
72  unsigned int retry_counter, state, connected = FALSE;
73  /*
74  void* signal(SIGALRM, alarmHandler);
75
76  void alarmHandler(){
77  retry_counter++;
78  prtinf("Alarm TRIGGERED, retry_counter = %d\n", retry_counter);
79  }
80  */
81  void switchC1(){
82      if (C1 == 0x00) C1 = 0x40; //Ns1
83      else            C1 = 0x00; //Ns0
84  }
85
86
87  int llopen(int port, char mode){
88
89      if(port != 0 && port != 1){
90          printf("ERROR:llopen: invalid serial port number: %d\n", port);
91          exit(-1);
92      }
93
94      if((mode != TRANSMITTER) && (mode != RECEIVER)){
95          printf("Usage: invalid mode: %d\n", mode);
96          exit(-1);
97      }
98
99      char portstring[] = "/dev/ttyS";
100     char *portnum;
101     if(port == 1) portnum = "1";
102     else portnum = "0";
103     char *serial_name = strcat(portstring, portnum);
104     printf("Serial port: %s\n", serial_name);
105
106     unsigned char set_message[5] = {FLAG, A, C_SET, A^C_SET, FLAG};
107     unsigned char byte;
108
109     int fd, res;
110     struct termios newtio;
111
112     /*
113     Open serial port dmevice for reading and writing and not as controlling tty
114     because we don't want to get killed if linenoise sends CTRL-C.
115     */
116
117     if((fd = open(serial_name, O_RDWR | O_NOCTTY )) < 0){
118         printf("llopen()::could not open serial port %d\n", port);
119         exit(-1);
120     }
121
122     if (fd < 0 ) {
123         perror(serial_name);
124         exit(-1);
125     }
```

```
126
127     if ( tcgetattr(fd,&oldtio) == -1) { /* save current port settings */
128         perror("tcgetattr");
129         exit(-1);
130     }
131
132     bzero(&newtio, sizeof(newtio));
133     newtio.c_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;
134     newtio.c_iflag = IGNPAR;
135     newtio.c_oflag = 0;
136
137     /* set input mode (non-canonical, no echo,...) */
138     newtio.c_lflag = 0;
139
140     newtio.c_cc[VTIME]    = 0;   /* inter-character timer unused */
141     newtio.c_cc[VMIN]     = 1;   /* blocking read until 5 chars received */
142
143     /*
144     VTIME e VMIN devem ser alterados de forma a proteger com um temporizador a
145     leitura do(s) pr□ximo(s) caracter(es)
146     */
147
148     tcflush(fd, TCIOFLUSH);
149
150     if ( tcsetattr(fd,TCSANOW,&newtio) == -1) {
151         perror("tcsetattr");
152         exit(-1);
153     }
154
155     //TRANSMITTER
156
157     if(mode == TRANSMITTER){
158
159         retry_counter = 0;
160         connected = 0;
161         state = START;
162
163         printf("Sending SET message...\n");
164
165         res = write(fd, set_message, sizeof(set_message));
166         printf("llopen:write: %d bytes written\n", res);
167
168         while(!connected){
169             if(state != END){
170                 printf("fd: %d | byte: %02x | sizeofbyte: %lu\n", fd, byte, sizeof(byte));
171                 if(read(fd, &byte, sizeof(byte)) == 0){
172                     printf("Nothing read from UA.\n");
173                 }
174                 printf("Current byte being proccessed: %02x\n", byte);
175             }
176
177             printf("Received State: %d\n", state);
178
179             switch(state){
180
181                 case START:
182                 if(byte == FLAG){
183                     state = FLAG_RCV;
184                     printf("UA First FLAG processed successfully: %02x\n", byte);
185                 }
186                 else { state = START; printf("UA START if 1\n"); }
187                 break;
188
```

```
189                 case FLAG_RCV:
190                 if(byte == A){
191                     state = A_RCV;
192                     printf("UA A processed successfully: %02x\n", byte);
193                 }
194                 else if(byte == FLAG) state = FLAG_RCV;
195                 else state = START;
196                 break;
197 
198                 case A_RCV:
199                 if(byte == UA){
200                     state = UA_RCV;
201                     printf("UA C_SET processed successfully: %02x\n", byte);
202                 }
203                 else if(byte == FLAG) state = FLAG_RCV;
204                 else state = START;
205                 break;
206 
207                 case UA_RCV:
208                 if(byte == (A ^ UA)){
209                     state = BCC_OK;
210                     printf("UA UA_RCV processed successfully: %02x\n", byte);
211                 }
212                 else if(byte == FLAG) state = START;
213                 else state = START;
214                 break;
215 
216                 case BCC_OK:
217                 if(byte == FLAG){
218                     state = END;
219                     printf("UA Last FLAG processed successfully: %02x\n", byte);
220                 }
221                 break;
222 
223                 case END:
224                 printf("UA processed successfully.\n");
225                 connected = TRUE;
226                 break;
227 
228                 default:
229                 printf("You shouldnt be here. Leave.\n");
230                 break;
231             }
232         }
233     }
234 
235     //RECEIVER
236 
237     else{
238 
239         state = START;
240         STOP = FALSE;
241 
242         /* state machine for SET message processing */
243         int j;
244         for(j=0; j<5; j++){
245             printf("START SET[%d]: %02x\n", j, set_message[j]);
246         }
247 
```

```
248         while(!STOP){
249
250             if(state != END){
251                 res = read(fd, &byte, sizeof(byte));
252                 printf("Current byte being proccessed: %02x\n", byte);
253                 printf("SET RECEIVE FD: %d\n", res);
254             }
255
256             switch(state){
257
258                 case START:
259                 if(byte == FLAG){
260                     state = FLAG_RCV;
261                     printf("First FLAG processed successfully: %02x\n", byte);
262                 }
263                 else { state = START; printf("START if 1\n"); }
264                 break;
265
266                 case FLAG_RCV:
267                 if(byte == A) {
268                     state = A_RCV;
269                     printf("A processed successfully: %02x\n", byte);
270                 }
271                 else if(byte == FLAG){ state = FLAG_RCV; printf("FLAG_RCV if 1\n"); }
272                 else{ state = START; printf("FLAG_RCV if 2\n"); }
273                 break;
274
275                 case A_RCV:
276                 if(byte == C_SET) {
277                     state = C_RCV;
278                     printf("C_SET processed successfully: %02x\n", byte);
279                 }
280                 else if(byte == FLAG){ state = FLAG_RCV; printf("A_RCV if 1\n"); }
281                 else{ state = START; printf("A_RCV if 2\n"); }
282                 break;
283
284                 case C_RCV:
285                 printf("\nProcessing C_RCV\n");
286                 printf("set_message[1]: %02x | set_message[2]: %02x\n", set_message[1], set_message[2]);
287                 printf("Byte: %02x | SET: %02x\n", byte, set_message[3]);
288                 if(byte == (set_message[1] ^ set_message[2])){
289                     state = BCC_OK;
290                     printf("BCC processed successfully: %02x\n", byte);
291                 }
292                 else if(byte == FLAG){ state = FLAG_RCV; printf("\nC_RCV if 1\n"); }
293                 else { state = START; printf("\nC_RCV if 2\n"); }
294                 break;
295
296                 case BCC_OK:
297                 if(byte == FLAG){
298                     state = END;
299                     printf("Last FLAG processed successfully: %02x\n", byte);
300                 }
301                 else { state = START; printf("BCC_OK if 1\n"); }
302                 break;
303
304                 case END:
305                 printf("Reached end of State Machine\n");
306                 STOP = TRUE;
307                 break;
308
309                 default:
310                 printf("You shouldnt be here. go away.\n");
311                 break;
```

```
312             }
313         }
314
315         printf("\nSET processed successfully, sending UA message:\n");
316
317         unsigned char ua_message[5];
318         ua_message[0] = FLAG;
319         ua_message[1] = A;
320         ua_message[2] = UA;
321         ua_message[3] = UA ^ A;
322         ua_message[4] = FLAG;
323
324         int k;
325         for(k=0; k < 5; k++){
326             printf("UA[%d]: %02x\n", k, ua_message[k]);
327         }
328
329         int wfd;
330         wfd = write(fd, ua_message, sizeof(ua_message));
331         printf("UA Write FD: %d\n", wfd);
332
333         printf("Connection established\n");
334         printf("Serial port: %d", fd);
335     }
336
337     return fd;
338 }
339
340 int llwrite(int fd, char *bufferer, int len){
341
342     //char* bufferer = malloc(3);
343
344     /*
345     bufferer[0] = 0x77; //D
346     bufferer[1] = 0x7d; //DATA
347     bufferer[2] = 0x55; //Dn
348     */
349     unsigned char BCC2 = calculateBCC2(bufferer, len);
350
351
352     int newSize = stuffing(bufferer, len);
353
354     char* frame_to_send = malloc(6 + newSize);
355
356     frame_to_send[0] = FLAG;
357     frame_to_send[1] = A;
358     frame_to_send[2] = C1;
359     frame_to_send[3] = C1^A; // BCC1
360     memcpy(frame_to_send+4,bufferer,newSize );
361     frame_to_send[4+newSize] = BCC2;
362     frame_to_send[4+newSize+1] = FLAG;
363
364     //send bufferer to llread
365     int ret = write(fd, frame_to_send, 6+newSize);
366
367     //Variables to Send If Alarm Gets Triggered
368     fdGlobal = fd;
369     frameGlobal = frame_to_send;
370     lenGlobal = 6+newSize;
371
372     printf("llwrite:write: %d bytes written\n", ret);
373
374     //wait for RR confirmation response
```

```
375     unsigned char byte;
376     signal(SIGALRM, handle);
377     int aux = 0;
378     alarm(2);
379     while(breakflag && aux == 0)
380     {
381         aux = read(fd, &byte, 1);
382     }
383     alarm(0);
384     breakflag = 3;
385     if(aux <= 0){
386         printf("Nothing read from llread.\n");
387         return -1;
388     }
389     printf("%x\n",byte );
390     if(byte == REJ)
391     {
392         return 1;
393     }
394 
395     printf("Response message received: %02x\n", byte);
396     return 0;
397 }
398 
399 
400 int llread(int fd, char *data){
401 
402     printf("\nReading I Frame...\n");
403 
404     state = START;
405     STOP = FALSE;
406 
407     char *buffer = malloc(3000);
408     unsigned char byte;
409     unsigned int size = 0;
410     unsigned int dataSize = 0;
411 
412     int tries = 0;
413 
414     signal(SIGALRM, receiverHandle);
415     alarm(2);
416 
417     while(!STOP){
418         if(state != END){
419             if(read(fd, &byte, sizeof(byte)) == 0){
420                 printf("Error: Nothing read from llread.\n");
421                 sleep(1);
422                 tries++;
423                 if(tries == 3)
424                 {
425                     printf("Connection Lost!");
426                     exit(1);
427                 }
428             }
429             //printf("Current byte being proccessed: %02x\n", byte);
430         }
431         switch(state){
432 
433             case START:
434             if(byte == FLAG){
435                 state = FLAG_RCV;
436                 buffer[size] = byte;
437                 //printf("[%d]th element of buffer: %02x\n", size, buffer[size]);
```

```
438                 size++;
439                 //printf("First FLAG processed successfully: %02x\n", byte);
440             }
441             else { state = START; //printf("START if 1\n");
442         }
443         break;
444
445         case FLAG_RCV:
446         if(byte == A) {
447             state = A_RCV;
448             buffer[size] = byte;
449             //printf("[%d]th element of buffer: %02x\n", size, buffer[size]);
450             size++;
451             // printf("A processed successfully: %02x\n", byte);
452         }
453         else if(byte == FLAG){ state = FLAG_RCV; //printf("FLAG_RCV if 1\n");
454     }
455     else{ state = START; //printf("FLAG_RCV if 2\n");
456 }
457 break;
458
459 case A_RCV:
460 if(byte == C1) {
461     state = C1_RCV;
462     buffer[size] = byte;
463     //printf("[%d]th element of buffer: %02x\n", size, buffer[size]);
464     size++;
465     //printf("C1 processed successfully: %02x\n", byte);
466 }
467 else if(byte == FLAG){ state = FLAG_RCV; //printf("A_RCV if 1\n");
468 }
469 else{ state = START; //printf("A_RCV if 2\n");
470 }
471 break;
472
473 case C1_RCV:
474 //printf("Processing C1_RCV\n");
475 if(byte == (C1^A)){
476     state = BCC1_OK;
477     buffer[size] = byte;
478     //printf("[%d]th element of buffer: %02x\n", size, buffer[size]);
479     size++;
480     //printf("BCC1 processed successfully: %02x\n", byte);
481 }
482 else if(byte == FLAG){ state = FLAG_RCV; //printf("\nC1_RCV if 1\n");
483 }
484 else { state = START; //printf("\nC1_RCV if 2\n");
485 }
486 break;
487
488 case BCC1_OK:
489 //printf("Processing BCC1_OK\n");
490 if(byte == FLAG){
491     state = END;
492     buffer[size] = byte;
493     //printf("[%d]th element of buffer: %02x\n", size, buffer[size]);
494     size++;
495     //printf("BCC1_OK processing failure: %02x\n", byte);
496 }
497 else {
498     state = DATA_PROCESSING;
499     buffer[size] = byte;
500     data[dataSize] = byte;
```

```
500         data[dataSize] = byte;
501         //printf("[%d]th element of buffer: %02x\n", size, buffer[size]);
502         size++;
503         dataSize++;
504         //printf("Starting to proccess Data from I Frame...\n");
505     }
506     break;
507
508     case DATA_PROCESSING:
509     if(byte == FLAG){
510         state = END;
511         buffer[size] = byte;
512         //printf("[%d]th element of buffer: %02x\n", size, buffer[size]);
513         size++;
514         //printf("Finished processing Data: %02x\n", byte);
515     }
516     else {
517         state = DATA_PROCESSING;
518         buffer[size] = byte;
519         data[dataSize] = byte;
520         //printf("[%d]th element of buffer: %02x\n", size, buffer[size]);
521         size++;
522         dataSize++;
523         //printf("Expected BCC2: %02x\n", BCC2);
524         //printf("Processing data...\n");
525     }
526     break;
527
528     case BCC2_OK:
529     if(byte == FLAG){
530         state = END;
531         buffer[size] = byte;
532         //printf("[%d]th element of buffer: %02x\n", size, buffer[size]);
533         size++;
534         //printf("BCC2 processed successfully: %02x\n", byte);
535     }
536     else{ state = START; //printf("Failed BCC2 processing: %02x\n", byte);
537     }
538     break;
539
540     case END:
541     //printf("Reached end of I Frame Processing State Machine\n");
542     STOP = TRUE;
543     break;
544     }
545     }
546     alarm(0);
547     unsigned int newdatasize = deStuffing(data, dataSize) - 1;
548
549     //Send RR confirmation packet if BCC2 is correct
550
551     unsigned char data_BCC2 = calculateBCC2(data, newdatasize);
552     //unsigned char data_BCC2 = 0xff;
553
554     //printf("data_BCC2: %02x\n", data_BCC2);
555
556     if(data_BCC2 == (unsigned char) buffer[size - 2]){
557         if(C1 == 0x00)
558         switchC1();
559
560         //printf("BCC2 processed successfully.\n");
561         int ret = write(fd, &RR, sizeof(byte));
562         printf("llread:RR: %d bytes written\n", ret);
563     }
```

```
564 else{
565     printf("Error in BCC2, sending REJ message.\n");
566     int ret = write(fd, &REJ, sizeof(byte));
567     printf("llread:REJ: %d bytes written\n", ret);
568     return -1;
569 }
570 return newdatasize;
571 }
572
573
574 int llclose(int fd, int flag){
575     unsigned char byte;
576     int res;
577     unsigned char set_message[5] = {FLAG, A, C_DISC, A^C_DISC, FLAG};
578     if(flag == 0) //transmitter
579     {
580         //enviar disc_pack
581         char* DISC = malloc(5*sizeof(char));
582         DISC[0] = FLAG;
583         DISC[1] = A;
584         DISC[2] = C_DISC;
585         DISC[3] = A ^ C_DISC;
586         DISC[4] = FLAG;
587
588         write(fd, DISC, 5);
589
590         state = START;
591         STOP = FALSE;
592
593         //receber o DISK do receiver
594         while(!STOP){char* DISC = malloc(5*sizeof(char));
595             DISC[0] = FLAG;
596             DISC[1] = A;
597             DISC[2] = C_DISC;
598             DISC[3] = A ^ C_DISC;
599             DISC[4] = FLAG;
600
601             write(fd, DISC, 5);
602             if(state != END){
603                 res = read(fd, &byte, sizeof(byte));
604                 printf("Current byte being proccessed: %02x\n", byte);
605                 printf("DISC RECEIVE FD: %d\n", res);
606             }
607
608             switch(state){
609
610                 case START:
611                 if(byte == FLAG){
612                     state = FLAG_RCV;
613                     printf("First FLAG processed successfully: %02x\n", byte);
614                 }
615                 else { state = START; printf("START if 1\n"); }
616                 break;
617
618                 case FLAG_RCV:
619                 if(byte == A) {
620                     state = A_RCV;
621                     printf("A processed successfully: %02x\n", byte);
622                 }
623                 else if(byte == FLAG){ state = FLAG_RCV; printf("FLAG_RCV if 1\n"); }
624                 else{ state = START; printf("FLAG_RCV if 2\n"); }
625                 break;
```

```
626
627             case A_RCV:
628             if(byte == C_DISC) {
629                 state = C_RCV;
630                 printf("C_DISC processed successfully: %02x\n", byte);
631             }
632             else if(byte == FLAG){ state = FLAG_RCV; printf("A_RCV if 1\n"); }
633             else{ state = START; printf("A_RCV if 2\n"); }
634             break;
635
636             case C_RCV:
637             printf("\nProcessing C_RCV\n");
638             printf("set_message[1]: %02x | set_message[2]: %02x\n", set_message[1], set_message[2]);
639             printf("Byte: %02x | SET: %02x\n", byte, set_message[3]);
640             if(byte == (set_message[1] ^ set_message[2])){
641                 state = BCC_OK;
642                 printf("BCC processed successfully: %02x\n", byte);
643             }
644             else if(byte == FLAG){ state = FLAG_RCV; printf("\nC_RCV if 1\n"); }
645             else { state = START; printf("\nC_RCV if 2\n"); }
646             break;
647
648             case BCC_OK:
649             if(byte == FLAG){
650                 state = END;
651                 printf("Last FLAG processed successfully: %02x\n", byte);
652             }
653             else { state = START; printf("BCC_OK if 1\n"); }
654             break;
655
656             case END:
657             printf("Reached end of State Machine\n");
658             STOP = TRUE;
659             break;
660
661             default:
662             printf("You shouldnt be here. go away.\n");
663             break;
664         }
665     }
666     //enviar UA
667     unsigned char ua_message[5];
668     ua_message[0] = FLAG;
669     ua_message[1] = A;
670     ua_message[2] = UA;
671     ua_message[3] = UA ^ A;
672     ua_message[4] = FLAG;
673
674     int k;
675     for(k=0; k < 5; k++){
676         printf("UA[%d]: %02x\n", k, ua_message[k]);
677     }
678     int wfd;
679     wfd = write(fd, ua_message, 5);
680 }else { //receiver
681     //receber o disc
682     while(!STOP){
683         if(state != END){
684             res = read(fd, &byte, sizeof(byte));
685             printf("Current byte being proccessed: %02x\n", byte);
686             printf("DISC RECEIVE FD: %d\n", res);
687         }
688
```

```
689         switch(state){
690
691             case START:
692             if(byte == FLAG){
693                 state = FLAG_RCV;
694                 printf("First FLAG processed successfully: %02x\n", byte);
695             }
696             else { state = START; printf("START if 1\n"); }
697             break;
698
699             case FLAG_RCV:
700             if(byte == A) {
701                 state = A_RCV;
702                 printf("A processed successfully: %02x\n", byte);
703             }
704             else if(byte == FLAG){ state = FLAG_RCV; printf("FLAG_RCV if 1\n"); }
705             else{ state = START; printf("FLAG_RCV if 2\n"); }
706             break;
707
708             case A_RCV:
709             if(byte == C_DISC) {
710                 state = C_RCV;
711                 printf("C_DISC processed successfully: %02x\n", byte);
712             }
713             else if(byte == FLAG){ state = FLAG_RCV; printf("A_RCV if 1\n"); }
714             else{ state = START; printf("A_RCV if 2\n"); }
715             break;
716
717             case C_RCV:
718             printf("\nProcessing C_RCV\n");
719             printf("set_message[1]: %02x | set_message[2]: %02x\n", set_message[1], set_message[2]);
720             printf("Byte: %02x | SET: %02x\n", byte, set_message[3]);
721             if(byte == (set_message[1] ^ set_message[2])){
722                 state = BCC_OK;
723                 printf("BCC processed successfully: %02x\n", byte);
724             }
725             else if(byte == FLAG){ state = FLAG_RCV; printf("\nC_RCV if 1\n"); }
726             else { state = START; printf("\nC_RCV if 2\n"); }
727             break;
728
729             case BCC_OK:
730             if(byte == FLAG){
731                 state = END;
732                 printf("Last FLAG processed successfully: %02x\n", byte);
733             }
734             else { state = START; printf("BCC_OK if 1\n"); }
735             break;
736
737             case END:
738             printf("Reached end of State Machine\n");
739             STOP = TRUE;
740             break;
741
742             default:
743             printf("You shouldnt be here. go away.\n");
744             break;
745         }
746     }
747     //enviar disc
748     char* DISC = malloc(5*sizeof(char));
749     DISC[0] = FLAG;
750     DISC[1] = A;
751     DISC[2] = C_DISC;
```

```
752         DISC[3] = A ^ C_DISC;
753         DISC[4] = FLAG;
754 
755         write(fd, DISC, 5);
756 
757         //Receber UA
758         while(!connected){
759             if(state != END){
760                 printf("fd: %d | byte: %02x | sizeofbyte: %lu\n", fd, byte, sizeof(byte));
761                 if(read(fd, &byte, sizeof(byte)) == 0){
762                     printf("Nothing read from UA.\n");
763                 }
764                 printf("Current byte being proccessed: %02x\n", byte);
765             }
766 
767             printf("Received State: %d\n", state);
768 
769             switch(state){
770 
771                 case START:
772                 if(byte == FLAG){
773                     state = FLAG_RCV;
774                     printf("UA First FLAG processed successfully: %02x\n", byte);
775                 }
776                 else { state = START; printf("UA START if 1\n"); }
777                 break;
778 
779                 case FLAG_RCV:
780                 if(byte == A){
781                     state = A_RCV;
782                     printf("UA A processed successfully: %02x\n", byte);
783                 }
784                 else if(byte == FLAG) state = FLAG_RCV;
785                 else state = START;
786                 break;
787 
788                 case A_RCV:
789                 if(byte == UA){
790                     state = UA_RCV;
791                     printf("UA C_SET processed successfully: %02x\n", byte);
792                 }
793                 else if(byte == FLAG) state = FLAG_RCV;
794                 else state = START;
795                 break;
796 
797                 case UA_RCV:
798                 if(byte == (A ^ UA)){
799                     state = BCC_OK;
800                     printf("UA UA_RCV processed successfully: %02x\n", byte);
801                 }
802                 else if(byte == FLAG) state = START;
803                 else state = START;
804                 break;
805 
806                 case BCC_OK:
807                 if(byte == FLAG){
808                     state = END;
809                     printf("UA Last FLAG processed successfully: %02x\n", byte);
810                 }
811                 break;
812 
813                 case END:
814                 printf("UA processed successfully.\n");
```

```
                    connected = TRUE;
                    break;

                    default:
                    printf("You shouldnt be here. Leave.\n");
                    break;
                }
            }
        }

        /*sleep(2);
        if ( tcsetattr(fd,TCSANOW,&oldtio) == -1) {
        perror("tcsetattr");
        return 1;
    }
    close(fd);*/

    return 0;
    }

    int stuffing(char * package, int length)
    {
        int size = length;
        int i;
        for(i = 0; i < length;i++)
        {
            char oct = package[i]; //oct means byte
            if(oct == FLAG || oct == ESC){
                size++;
            }
        }
        if(size == length) //same size no need to stuff
        return size;

        for(i = 0; i < size; i++)
        {
            char oct = package[i];
            if(oct == FLAG || oct == ESC)
            {
                memmove(package + i + 2, package + i+1, size - i); //moving everything to the front
                if (oct == FLAG)
                {
                    package[i+1] = XOR_7E_20;
                    package[i] = ESC ;
                }
                else package[i+1] = XOR_7D_20;
            }
        }
        return size; //return the new size of the package*/
    }

    int deStuffing(char * package, int length){

        int size = length;
        int i;
        for(i = 0; i < size; i++)
        {
            char oct = package[i]; //oct means byte
            if(oct == ESC)
            {
                if(package[i+1] == XOR_7E_20){
                    package[i] = FLAG; //7E means that was a previous byte flag in there
                    memmove(package + i + 1, package + i + 2,length - i + 2);
```

application_layer.h

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>

#define TO_READ "r"
#define TO_WRITE "w"

#define DATA_PACK 1
#define START_PACK 2
#define END_PACK 3
#define TSIZE 0
#define TNAME 1

#define PACK_SIZE 600

int transmitter(char * fileName, int fd);
int receiver(int fd);
int open_file(FILE ** file, char * fileName);
unsigned long file_size(FILE * file,  int * fileSize);
int create_start_end_package(int type, char * fileName,  int size, char * package);
int check_num_bytes(int size);
int get_file_info(char* buffer, int buffsize,  int *size, char *name);
int create_file(FILE ** file, char * fileName);
int get_data (char * buffer, int size);
int create_data_package(char *buffer, int size, char packageID);

```

application_layer.c

```c
#include "application_layer.h"
#include "link_layer.h"

//static int mode = 0; //different modes for different data
//static int PACK_SIZE = 0;
//static int TRAMA_SIZE = 0;
static int DEBUG_FLAG = 0;
static int numRetries = 0;
unsigned char packNum = 0;

int transmitter(char * fileName, int fd) //envio da trama com SET
{
  FILE * file = NULL;
  //openfile
   int fileSize = 0;
  DEBUG_FLAG = open_file(&file, fileName);
  if(DEBUG_FLAG != 0)
    return -1;
  unsigned long size = file_size(file, &fileSize);

  //criar uma package com SET
  //receiver envia mensagem!
  //depois avançar
  //enviar mensagem com o START_PACK (2) END_PACK(3)
  char * packStart = malloc(1024);
  int packageSize = create_start_end_package(START_PACK, fileName, size, packStart);
  llwrite(fd, packStart, packageSize);      //int fd, char *buffer, int len);
  free(packStart);

  //SEND FILE
  char *data = malloc(1024);
  //char count = get_sequence_number(); // TODO::controloooo
  char packCount = 0;
  int bytesWritten = 0;
  int writeInt = 0;
  int aux = 0;
  int res = 0;
  int bytesRead = 0;
  while(bytesWritten < size && numRetries < MAXRETRIES)
  {

    fseek(file, bytesWritten, SEEK_SET);
      res = fread(data, 1, PACK_SIZE, file);
      printf("res: %d\n", res);
      bytesRead = res;

      res = create_data_package(data, res, packCount);


    writeInt = llwrite(fd, data, res);
    if(writeInt == 0)
    {
    packCount++;
      packCount %= 255; //caso ultrupasse os 255bytes disponíveis do count
      bytesWritten += bytesRead;
      aux = 0;
      //count ^= 1; aqui incrementava-se
    } else if(writeInt != 0){
      numRetries++;
      printf("CONECTION LOST\n");
      aux = 1;

    }
  }
  if(numRetries == 3)
  {
```

```
    numRetries = 0;
    free(data);
    exit(1);
  }
  numRetries = 0;
  printf("ENVIADO TUDO");
  free(data);


  //FINALIZAR
  char * end = malloc(1024);
  packageSize = create_start_end_package(END_PACK, fileName, size, end);

  if(llwrite(fd, end, packageSize) != 0)
  {
    printf("Unable to send END PACKAGE\n" );
    free(end);
    exit(1);
  }
  free(end);

  //ENVIAR O DISC E FECHAR
  if(file != NULL)
  fclose(file);
  else{
      printf("File NULL\n");
  }

  return 0;
}

int receiver(int fd){
  //RECEIVER

  //receive START signal
  char *start = malloc(1024);
  int startSize = llread(fd, start);
  int fileSize;
  char *name = malloc(1024);
  if( get_file_info(start, startSize, &fileSize, name) == -1)
  {
    printf("Error reading start package\n");
    exit(1);
  }
  printf("FILEZISE: %x\n", fileSize);
  FILE *file = NULL;

  if(create_file(&file, name) != 0)
    printf("not opened file\n");
  printf("opened file\n");
  free(name);

  //receber ficheiro e gravar
  int bytesRead = 0;
  char * buffer = malloc(1024);
  int checkRead = 0;
  while(bytesRead<fileSize)
  {
    int size;
    checkRead = llread(fd, buffer);
    if(checkRead == -1)
    {
      printf("Retrying Reading the same package\n");
```

```
130     } else {
131       size = get_data(buffer, size);
132       printf("Size:%d --- %d\n",bytesRead, size);
133       if(size!= -1 && getRR() == 0)
134       {
135         fwrite(buffer, 1, size, file);
136         bytesRead += size;
137       }
138       else if(size != 0 && getRR() == 1)
139       {
140         fseek(file,bytesRead, SEEK_SET);
141         fwrite(buffer, 1, size, file);
142         bytesRead += size;
143         setRR();
144       }
145       else{ printf("Size is negative'-'\n");}
146     }
147   }
148   free(buffer);
149   if(bytesRead != fileSize)
150     printf("Wrong Number Bytes\n");
151   printf("Enviado Direito\n");
152
153   //Finalizar
154   char *end = malloc(1024);
155   int endSize = llread(fd, end);
156   name = malloc(1);
157   if(get_file_info(end, endSize, &fileSize, name) == -1)
158   {
159     printf("Error reading end package\n");
160     exit(1);
161   }
162   printf("\nEnd Read name: %s - size: %d \n", name, fileSize);
163   free(name);
164
165 return 0;
166 }
167
168 int open_file(FILE ** file, char * fileName)
169 {
170   *file = fopen(fileName, "r+");
171   if(*file == NULL )
172   {
173     printf("File does not exist");
174     return -1;
175   }
176
177   return 0;
178 }
179
180 int create_file(FILE ** file, char * fileName)
181 {
182   *file = fopen(fileName, "w+");
183   if(*file == NULL )
184   {
185     printf("File does not exist");
186     return -1;
187   }
188
189   return 0;
190 }
191
192 unsigned long file_size(FILE * file,  int * fileSize)
193 {
```

```
194   unsigned long size = 0;
195   int fd = fileno(file);
196   lseek(fd, 0L, SEEK_END); // seek to end of file
197
198   size = ( int) ftell(file);   // get current file pointer
199   lseek(fd, 0L, SEEK_SET); // seek back to beginning of file
200                            // proceed with allocating memory and reading the file
201   if(size <= 0)
202   {
203     printf("File size is 0 or less");
204     return -1;
205   }
206   return size;
207 }
208
209 int create_start_end_package(int type, char * fileName,  int size, char * package)
210 {
211   //size = 10968
212   int nameLength = strlen(fileName);
213   int sizeLength = check_num_bytes(size);
214
215   int packSize = 5 + nameLength + sizeLength;
216   //package = realloc(package, packSize);
217
218   package[0] = type;
219   package[1] = TSIZE;
220   package[2] = (char) sizeLength;
221
222   int i = 3;
223   while(size != 0){
224       package[i] = (unsigned char) size;
225       i++;
226       size >>=8;
227   } //size is written backwards
228
229   package[i] = TNAME;
230   ++i;
231   package[i] = (char) nameLength;
232   ++i;
233
234   int j;
235   for(j = 0; j < nameLength; i++, j++)
236     package[i] = fileName[j];
237
238   return packSize;
239 }
240
241 int check_num_bytes(int size)
242 {
243   int count = 0;
244   while(size != 0)
245   {
246     size >>= 8;
247     count++;
248   }
249   return count;
250 }
251
252 int get_file_info(char* buffer, int buffsize,  int *size, char *name)
253 {
254   printf("%d\n", buffsize);
255   int fileSize = 0;
256   int i=0; //START
257   if(buffer[i] != START_PACK && buffer[i] != END_PACK)
258   {
```

```
259       printf("%x\n", buffer[i]);
260       printf("get file info: buffer error\n");
261       return -1;
262     }
263     i++; //TSIZE
264     if(buffer[i] != TSIZE)
265     {
266       printf("get file info: tsize\n");
267       return -1;
268     }
269
270     i++; //numBytesSize - T
271     int sizeLength = (int) buffer[i];
272     i++; //Numero de Bytes do ficheiro - L
273     int j;
274     for(j=0; j< sizeLength; j++, i++)
275     {
276       int k;
277       unsigned char ch = (unsigned char) buffer[i];
278       unsigned int curr = (unsigned int) ch;
279       for(k = 0; k < j; k++)
280           curr = curr << 8;
281       fileSize += curr;
282     }
283     *size = fileSize;
284     printf("Tamanho do ficheiro %d\n", *size);
285
286     //i++; //TNAME
287     printf("buffer[%d]=%02x\n", i, buffer[i]);
288
289     if(buffer[i] != TNAME)
290       return -1;
291     i++; //NAME size
292     int nameLength = buffer[i];
293     printf("name length: %02x\n", buffer[i]);
294     i++; //NAME data
295     //name = realloc(name, nameLength);
296     printf("Nome do ficheiro:\n");
297     for(j=0; j<nameLength; j++, i++)
298     {
299       name[j] = buffer[i];
300       printf("%d", name[j]);
301     }
302     name[j] = '\0';
303
304     return 0;
305   }
306
307   int create_data_package(char *buffer, int size, char packageID)
308   {
309     int length = size + 4;
310     char * copy = malloc(size); //buffer para adicionar depois
311     memcpy(copy, buffer, size);
312   //  buffer = realloc(buffer, length);
313     buffer[0] = DATA_PACK; // 1
314     buffer[1] = (unsigned char) packageID; //num sequencia
315     buffer[2] = (unsigned char) (size / 256); //num bytes
316     buffer[3] = (unsigned char) (size % 256); //restantes bytes
317
318     memcpy(buffer+4, copy, size);
319
320     free (copy);
321
```

```
322     return length;
323   }
324
325   int get_data (char * buffer, int size)
326   {
327     printf("%d\n", size);
328     if(buffer[0] != DATA_PACK)
329       return -1;
330     if((unsigned char) buffer[1] != packNum)
331       return -1;
332
333     packNum = (packNum +1) % 255; //incrementar e ficar no máximo em 255bytes
334
335     unsigned char l1 = (unsigned char) buffer[2];
336     unsigned char l2 = (unsigned char) buffer[3];
337     int length = (int) (256*l1 + l2); //nos slides, numBytes
338     char *copy = malloc(length);
339
340     memcpy(copy, buffer + 4, length);
341     memcpy(buffer, copy, length);
342
343     free(copy);
344     return length;
345   }
346
```

main.c

```c
/*Non-Canonical Input Processing*/

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

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

int main(int argc, char** argv) {


if(strcmp(argv[2], "TRANSMITTER") == 0){
    if (argc != 4) {
        printf("ERROR: Wrong number of arguments.\n");
        exit(0);
    }
  int t_fd;
    char *t_buf = malloc(sizeof(*t_buf));

    t_fd = llopen((*argv[1])-'0', 0);
    printf("\nmain.c: Transmitter: descriptor after llopen: %d\n", t_fd);
  transmitter(argv[3], t_fd); //nome do ficheiro
  llclose(t_fd, 0); //0 significa o transmitter

}

else if(strcmp(argv[2], "RECEIVER") == 0) {

    int r_fd;
    char *r_buf = malloc(sizeof(*r_buf));


    r_fd = llopen((*argv[1])-'0', 1);
    receiver(r_fd);
    printf("\nmain.c: Receiver: descriptor after llopen: %d\n", r_fd);
    /*
    if(llread(r_fd, r_buf) == 0){
        printf("\nError: main.c: Receiver: Nothing to read from llread.\n");
    }
    */
    free(r_buf);
    llclose(r_fd, 1); //1 é o RECEIVER

}

else{
    printf("\nERROR: Invalid argument provided: %s\n", argv[2]);
    exit(-1);
}

    return 0;
}

```

Makefile

```
#!Transmit an image .gif through the serial port
serialCom: main.c link_layer.c link_layer.h application_layer.c application_layer.h
	gcc -Wall main.c link_layer.c application_layer.c -o serialCom

debug: main.c link_layer.c link_layer.h
	gcc -g -Wall main.c link_layer.c application_layer.c -o serialCom

clean:
	rm -f serialCom

```