

**3ºano – MIEIC – Outubro 2017**

**Protocolo de Ligação de Dados**

**ESOF**

**Turma 2 - Grupo y**

**João Francisco Veríssimo Dias Esteves – up201505145**

**João Miguel Matos Monteiro – up201506130**

**Maria Eduarda Santos Cunha – up201506524**

Índice

[1. Sumário 4](#_Toc496979744)

[2. Introdução 5](#_Toc496979745)

[3. Arquitetura e Estrutura do Código 6](#_Toc496979746)

[3.1. Camada de Ligação de Dados 6](#_Toc496979747)

[3.2. Camada de Aplicação 6](#_Toc496979748)

[3.3. Interface 6](#_Toc496979749)

[4. Casos de Uso Principais 7](#_Toc496979750)

[5. Protocolo de Ligação Lógica 8](#_Toc496979751)

[5.1. llopen() e llclose() 8](#_Toc496979752)

[5.2. llwrite() e llread() 8](#_Toc496979753)

[6. Protocolo de Aplicação 9](#_Toc496979754)

[7. Validação 10](#_Toc496979755)

[8. Elementos de Valorização 11](#_Toc496979756)

[9. Conclusão 12](#_Toc496979757)

[10. Anexos 13](#_Toc496979758)

# Sumário

Este projeto consiste numa aplicação capaz de transmitir ficheiros entre 2 computadores pelo uso de uma porta de série assíncrona, resistente a certas falhas que possam surgir durante o processo de envio, nomeadamente a introdução de erros através do fio disponível na porta de série e o fecho da mesma.

O problema proposto foi implementado com sucesso e absolutamente essencial para a consolidação dos conceitos lecionados nas aulas teóricas e laboratoriais.

# Introdução

Este relatório tem como objetivo complementar o primeiro projeto da Unidade Curricular Redes de Computadores, intitulado “Protocolo de Ligação de Dados”, com vista a permitir uma análise do código com o auxílio da perspetiva de quem o escreveu.

A funcionalidade principal desse projeto é permitir a comunicação de dados fiável entre 2 computadores ligados por um cabo de série.

O relatório encontra-se dividido nas seguintes secções:

**Arquitetura:** Blocos funcionais e interface;

**Estrutura do Código:** APIs, principais estruturas de dados, funções e respeitante relação com a arquitetura;

**Casos de Uso Principais:** Respetiva identificação e sequências de chamadas de funções;

**Protocolo de Ligação Lógica:** Descrição da estratégia aplicada e identificação dos aspetos funcionais principais;

**Protocolo de Aplicação:** Semelhante ao encontrado no Protocolo de Ligação Lógica, mas para a Aplicação;

**Validação:** Testes efetuados e resultados;

**Elementos de Valorização:** Componentes adicionais passíveis de serem implementadas.

# Arquitetura e Estrutura do Código

O projeto encontra-se dividido em 2 camadas, a de aplicação e a de ligação de dados, estando cada uma implementada na sua header, *appAPI.h* e *llAPI.h*. O projeto apresenta ainda os ficheiros *writenoncanonical.c* e *noncanonical.c* para a interface com o utilizador.

## Camada de Ligação de Dados

É implementada pelo ficheiro *llAPI.h*, sendo usada para transmitir um pacote de um transmissor para um recetor de uma forma segura em termos de falhas de ligação.

A camada de ligação de dados usa a função *llopen()* para estabelecer a ligação, *llwrite()* para transmitir pacote a pacote, *llread()* para receber pacote a pacote e *llclose()* para terminar a ligação. Há uma variável global, *timedOut*, que é usada para assinalar o timeout em cada envio de trama. Para a leitura de uma trama, é usada a função *readFrame()* que preenche a trama lida num parâmetro e retorna o tipo de trama lido através do *enum FrameTypeRes*. O *enum CommsType* é usado como parâmetro da *llopen()* para diferenciar entre transmissor e recetor.

São usadas várias macros para definir valores, tais como C\_UA para o byte de controlo da trama UA, na camada de ligação de dados, e T\_FILE\_SIZE para indicar que os dados da correspondente estrutura TLV, num pacote de controlo da camada de aplicação, correspondem ao tamanho em bytes do ficheiro em transmissão.

## Camada de Aplicação

É implementada pelo ficheiro *appAPI.h*, sendo usada, pelo transmissor, para ler um ficheiro e transmiti-lo pacote a pacote, de forma segura, pela camada de ligação de dados e, pelo recetor, para receber um ficheiro pacote a pacote, de forma segura, pela camada de ligação de dados. Esta camada apresenta, ainda, o progresso em percentagem enquanto é efetuada a transmissão e um relatório final com os bytes transmitidos/recebidos, os bytes que o ficheiro original tem, o tempo total gasto e o tempo gasto apenas durante a transferência.

A camada de aplicação tem como pontos de entrada as funções *appWrite()* para o transmissor e *appRead()* para o recetor. Os seus argumentos são a porta série a utilizar e, no caso do transmissor, o nome do ficheiro a enviar. A criação do ficheiro recebido é feita pela função *writeLocalFile().* Os relatórios finais usam a *struct ExecTimes*, que contém *struct*s *timeval* para os tempos de início do programa, início da transferência, fim da transferência e fim do programa.

## Interface

É implementada pelos ficheiros *writenoncanonical.c* para o transmissor e *noncanonical.c* para o recetor, que obtêm como argumentos a porta série a usar e, no caso do transmissor, o nome do ficheiro a transmitir. A interface passa estes dados para a camada de aplicação.

# Casos de Uso Principais

Identificação

Sequência de chamada de funções

# Protocolo de Ligação Lógica

Identificação dos principais aspetos funcionais

Descrição da estratégia de implementação destes aspetos com a apresentação de extratos de código.

## llopen() e llclose()

A solução pode ser dividida em três fases.

Através do algoritmo de Pesquisa Exata ou Aproximada, o utilizador insere o nome da rua que está à procura e verifica se existe.

## llwrite() e llread()

A solução pode ser dividida em três fases.

Através do algoritmo de Pesquisa Exata ou Aproximada, o utilizador insere o nome da rua que está à procura e verifica se existe.

# Protocolo de Aplicação

Identificação dos principais aspetos funcionais

Descrição da estratégia de implementação destes aspetos com a apresentação de extratos de codigo

# Validação

De forma a testar a eficiência do código implementado, procedemos à realização de vários testes, como a transferência do ficheiro sem qualquer tipo de “obstáculo” (fig.x), fechando e voltando a abrir a porta de série (fig.y), fechando-a até ao TIMEOUT (fig.z) e com a introdução de erros (fig.w).

Para os diferentes tipos de teste, utilizou-se vários ficheiros: pinguim.gif e alguns ficheiros extra, de entre os quais ficheiros do tipo .txt e imagens com diferentes tamanhos mais pesados que o sugerido.

Como resposta a estes testes, é imprimido na consola o progresso em percentagem do envio do ficheiro, se foi ou não enviado com sucesso, os TIMOUTS, se deu reject etc.

Eficiência do protocolo de ligação de dados

caraterização estatística da eficiência do protocolo, feita com recurso a medidas sobre o código desenvolvido. A caracterização teórica de um protocolo Stop&Wait, que deverá ser usada como termo de comparação, encontra-se descrita nos slides de Ligação Lógica das aulas teóricas

# Elementos de Valorização

**Seleção do parâmetro ficheiro pelo utilizador:** Aquando da chamada da função do emissor, o ficheiro a ser enviado é um dos parâmetros.

**Implementação de REJ:**

**Registo de ocorrências:**

# Conclusão

A realização deste projeto, ainda que muito trabalhosa, foi fundamental para a interiorização dos conceitos lecionados nas aulas teóricas e laboratoriais.

Consideramos que o nosso objetivo foi cumprido, dado que respondemos com sucesso a todas as especificações pedidas no enunciado do trabalho. Temos 2 camadas independentes, mas com uma ligação unidirecional, já que a aplicação exerce controlo sobre a ligação de dados.

Após alguma reflexão, concluímos que, se nos fosse concedido mais tempo, podíamos implementar geração aleatória de erros, a receção na linha de comandos de parâmetros como baud rate, o valor de *time out*, o número de tentativas, o tamanho máximo dos pacotes, etc.

# Anexos

appAPI.h

1. #include "llAPI.h"
2. #include <stdio.h>
3. #include <math.h>
4. #include <sys/time.h>
5. #define C\_APP 0
6. #define TLV\_T 0
7. #define TLV\_L 1
8. #define TLV\_V 2
9. #define DATA\_N 1
10. #define DATA\_L2 2
11. #define DATA\_L1 3
12. #define DATA\_P1 4
13. #define T\_FILE\_SIZE 0
14. #define T\_FILE\_NAME 1
15. #define DATA\_PACKET 1
16. #define START\_PACKET 2
17. #define END\_PACKET 3
18. struct ExecTimes {
19. struct timeval \*startTime;
20. struct timeval \*startDataTime;
21. struct timeval \*endDataTime;
22. struct timeval \*endTime;
23. };
24. void readFileSize(char \*fileSizeChars, int \*fileLength, int arrayLength) {
25. \*fileLength = 0;
26. int hexOrder = 0;
27. for (int i = arrayLength - 1; i >= 0; i--) {
28. \*fileLength += (fileSizeChars[i] & 0xF) \* pow(16, hexOrder++);
29. \*fileLength += ((fileSizeChars[i] & 0xF0) >> 4) \* pow(16, hexOrder);
30. hexOrder++;
31. }
32. }
33. /\*\*
34. Reads data in packet to fileBuffer sequentially, reallocating it.
35. \*/
36. int processDataPacket(char \*packet, char \*\*fileBuffer, int \*fileBufferLength,
37. int \*seqNumMismatches) {
38. static int prevSeqNum = -1;
39. unsigned char sequenceNumber = packet[DATA\_N];
40. if (prevSeqNum == -1) {
41. prevSeqNum = sequenceNumber;
42. } else if ((prevSeqNum + 1) % 255 != sequenceNumber) {
43. printf("Warning: Sequence number mismatch in data packet.\n");
44. (\*seqNumMismatches)++;
45. #ifdef DEBUG
46. printf("prevSeqNum = %d | sequenceNumber = %d\n",
47. prevSeqNum, sequenceNumber);
48. #endif
49. }
50. int dataSize = 256 \* (unsigned char) packet[DATA\_L2]
51. +(unsigned char) packet[DATA\_L1];
52. \*fileBuffer = realloc(\*fileBuffer, \*fileBufferLength + dataSize);
53. for (int i = 0; i < dataSize; i++) {
54. (\*fileBuffer)[\*fileBufferLength + i] = packet[DATA\_P1 + i];
55. }
56. \*fileBufferLength += dataSize;
57. prevSeqNum = sequenceNumber;
58. return 0;
59. }
60. /\*\*
61. Reads file length and filename from packet, if they exist.
62. \*/
63. int processStartPacket(char \*packet, int packetLength, int \*fileLength, char \*\*filename)
64. {
65. bool setName = false, setSize = false;
66. int bytesRead = 1;
67. while (bytesRead < packetLength) {
68. int vLength = packet[bytesRead + TLV\_L];
69. switch (packet[bytesRead + TLV\_T]) {
70. case T\_FILE\_SIZE:
71. if (setSize) {
72. break;
73. }
74. char \*fileSizeChars = malloc(vLength + 1);
75. memcpy(fileSizeChars, packet + bytesRead + TLV\_V, vLength);
76. fileSizeChars[vLength] = '\0';
77. readFileSize(fileSizeChars, fileLength, vLength);
78. free(fileSizeChars);
79. setSize = true;
80. break;
81. case T\_FILE\_NAME:
82. if (setName) {
83. break;
84. }
85. \*filename = malloc(vLength);
86. if (filename == NULL) {
87. perror("processStartPacket - malloc");
88. return -1;
89. }
90. memcpy(\*filename, packet + bytesRead + TLV\_V, vLength);
91. setName = true;
92. break;
93. }
94. bytesRead += 2 + vLength;
95. }
96. if (!setSize) {
97. printf("processControlPacket(): Start packet did not contain file size.\n");
98. return -1;
99. }
100. return 0;
101. }
102. int processEndPacket(char \*endPacket, char \*startPacket, int packetLength) {
103. for (int i = 1; i < packetLength; i++) {
104. if (endPacket[i] != startPacket[i]) {
105. printf("processEndPacket(): End packet does not match start packet.\n");
106. return -1;
107. }
108. }
109. return 0;
110. }
111. int writeLocalFile(char \*filename, char \*fileBuffer, int fileBufferLength) {
112. FILE \*fp = fopen(filename, "wb");
113. if (fp == NULL) {
114. perror("writeLocalFile - fopen");
115. return -1;
116. }
117. if (fwrite(fileBuffer, 1, fileBufferLength, fp) == -1) {
118. perror("writeLocalFile - write");
119. fclose(fp);
120. return -1;
121. }
122. if (fclose(fp) == EOF) {
123. perror("writeLocalFile - fclose");
124. return -1;
125. }
126. return 0;
127. }
128. void printReceiverReport(int receivedBytes, int originalFileSize,
129. int seqNumMismatches, struct ExecTimes \*times) {
130. time\_t totalSeconds = times->endTime->tv\_sec - times->startTime->tv\_sec;
131. suseconds\_t totalMicroseconds;
132. if (times->endTime->tv\_usec > times->startTime->tv\_usec) {
133. totalMicroseconds = times->endTime->tv\_usec - times->startTime->tv\_usec;
134. } else {
135. totalMicroseconds = times->startTime->tv\_usec - times->endTime->tv\_usec;
136. }
137. time\_t dataSeconds = times->endDataTime->tv\_sec
138. - times->startDataTime->tv\_sec;
139. suseconds\_t dataMicroseconds;
140. if (times->endDataTime->tv\_usec > times->startDataTime->tv\_usec) {
141. dataMicroseconds = times->endDataTime->tv\_usec
142. - times->startDataTime->tv\_usec;
143. } else {
144. dataMicroseconds = times->startDataTime->tv\_usec
145. - times->endDataTime->tv\_usec;
146. }
147. printf("Received bytes: %d out of %d.\n", receivedBytes, originalFileSize);
148. printf("Sequence number mismatches: %d.\n", seqNumMismatches);
149. printf("Data transfer time: %.3fs.\n", dataSeconds + (double) dataMicroseconds
150. / pow(10, 6));
151. printf("Total time: %.3fs.\n", totalSeconds + (double) totalMicroseconds
152. / pow(10, 6));
153. }
154. int appRead(char port[]) {
155. struct timeval startTime;
156. if (gettimeofday(&startTime, NULL) == -1) {
157. perror("appWrite - startup gettimeofday");
158. }
159. char \*filename = NULL;
160. char \*fileBuffer = NULL, \*packet = NULL, \*startPacket = NULL;
161. int fileBufferLength = 0;
162. int fileLength = 0;
163. int packetLength = 0;
164. bool finished = false;
165. int seqNumMismatches = 0;
166. int fd = llopen(port, RECEIVER);
167. struct timeval startDataTime;
168. if (gettimeofday(&startDataTime, NULL) == -1) {
169. perror("appWrite - startup gettimeofday");
170. }
171. while (!finished) {
172. if ((packetLength = llread(fd, &packet)) == -1) {
173. printf("appRead(): llread() failed\n");
174. return -1;
175. }
176. if (packet == NULL) {
177. continue;
178. }
179. switch (packet[C\_APP]) {
180. case DATA\_PACKET:
181. processDataPacket(packet, &fileBuffer, &fileBufferLength,
182. &seqNumMismatches);
183. printf("Data received: %.2f%%\n", (double) fileBufferLength / fileLength \* 100);
184. break;
185. case START\_PACKET:
186. if (processStartPacket(packet, packetLength, &fileLength, &filename)
187. == -1) {
188. free(packet);
189. return -1;
190. }
191. startPacket = malloc(packetLength);
192. memcpy(startPacket, packet, packetLength);
193. break;
194. case END\_PACKET:
195. if (processEndPacket(packet, startPacket, packetLength) == -1) {
196. printf("appRead(): processEndPacket failed.\n");
197. free(packet);
198. free(fileBuffer);
199. return -1;
200. }
201. finished = true;
202. break;
203. }
204. free(packet);
205. }
206. free(startPacket);
207. struct timeval endDataTime;
208. if (gettimeofday(&endDataTime, NULL) == -1) {
209. perror("appWrite - startup gettimeofday");
210. }
211. if (llclose(fd) == -1) {
212. printf("appRead(): llclose() failed\n");
213. free(filename);
214. free(fileBuffer);
215. return -1;
216. }
217. if (writeLocalFile(filename, fileBuffer, fileBufferLength) == -1) {
218. printf("appRead(): writeLocalFile() failed.\n");
219. free(filename);
220. free(fileBuffer);
221. return -1;
222. }
223. free(filename);
224. free(fileBuffer);
225. struct timeval endTime;
226. if (gettimeofday(&endTime, NULL) == -1) {
227. perror("appWrite - startup gettimeofday");
228. }
229. struct ExecTimes times = { &startTime, &startDataTime, &endDataTime,
230. &endTime };
231. printf("\n\n");
232. printReceiverReport(fileBufferLength, fileLength, seqNumMismatches, &times);
233. return 0;
234. }
235. void printTransmitterReport(int bytesSent, int fileSize,
236. struct ExecTimes \*times) {
237. time\_t totalSeconds = times->endTime->tv\_sec - times->startTime->tv\_sec;
238. suseconds\_t totalMicroseconds;
239. if (times->endTime->tv\_usec > times->startTime->tv\_usec) {
240. totalMicroseconds = times->endTime->tv\_usec - times->startTime->tv\_usec;
241. } else {
242. totalMicroseconds = times->startTime->tv\_usec - times->endTime->tv\_usec;
243. }
244. time\_t dataSeconds = times->endDataTime->tv\_sec
245. - times->startDataTime->tv\_sec;
246. suseconds\_t dataMicroseconds;
247. if (times->endDataTime->tv\_usec > times->startDataTime->tv\_usec) {
248. dataMicroseconds = times->endDataTime->tv\_usec
249. - times->startDataTime->tv\_usec;
250. } else {
251. dataMicroseconds = times->startDataTime->tv\_usec
252. - times->endDataTime->tv\_usec;
253. }
254. printf("Transmitted bytes: %d out of %d.\n", bytesSent, fileSize);
255. printf("Data transfer time: %.3fs.\n",
256. dataSeconds + (double) dataMicroseconds / pow(10, 6));
257. printf("Total time: %.3fs.\n",
258. totalSeconds + (double) totalMicroseconds / pow(10, 6));
259. }
260. int appWrite(char port[], char filename[]) {
261. struct timeval startTime;
262. if (gettimeofday(&startTime, NULL) == -1) {
263. perror("appWrite - startup gettimeofday");
264. }
265. int portFd = llopen(port, TRANSMITTER);
266. if (portFd == -1) {
267. printf("appWrite(): Failed to open connection.\n");
268. return -1;
269. }
270. int fileSize = -1;
271. FILE \*fp = fopen(filename, "rb");
272. if (fp == NULL) {
273. perror("appWrite - fopen");
274. return -1;
275. }
276. struct stat statBuf;
277. stat(filename, &statBuf);
278. fileSize = statBuf.st\_size;
279. char startPacket[9 + strlen(filename)];
280. bzero(startPacket, 9 + strlen(filename));
281. startPacket[0] = START\_PACKET;
282. startPacket[1] = 0;
283. startPacket[2] = 4;
284. startPacket[3] = (fileSize & 0xFF000000) >> 24;
285. startPacket[4] = (fileSize & 0x00FF0000) >> 16;
286. startPacket[5] = (fileSize & 0x0000FF00) >> 8;
287. startPacket[6] = (fileSize & 0x000000FF);
288. startPacket[7] = 1;
289. startPacket[8] = strlen(filename);
290. memcpy(startPacket + 9, filename, strlen(filename));
291. if (llwrite(portFd, startPacket, 9 + strlen(filename)) == -1) {
292. #ifdef DEBUG
293. printf("appWrite(): Failed to send start packet.\n");
294. #endif
295. fclose(fp);
296. return -1;
297. }
298. struct timeval startDataTime;
299. if (gettimeofday(&startDataTime, NULL) == -1) {
300. perror("appWrite - startup gettimeofday");
301. }
302. char buffer[1024];
303. char n = 0;
304. int bytesRead = -2;
305. int totalBytesWritten = 0;
306. while (bytesRead = fread(buffer, 1, 1024, fp)) {
307. char dataPacket[bytesRead + 4];
308. bzero(dataPacket, bytesRead + 4);
309. dataPacket[0] = DATA\_PACKET;
310. dataPacket[1] = n % 255;
311. dataPacket[2] = bytesRead / 256;
312. dataPacket[3] = bytesRead % 256;
313. memcpy(dataPacket + 4, buffer, bytesRead);
314. int res = -2;
315. if ((res = llwrite(portFd, dataPacket, bytesRead + 4)) == -1) {
316. #ifdef DEBUG
317. printf("appWrite(): Failed to send data packet.\n");
318. #endif
319. fclose(fp);
320. return -1;
321. }
322. if (res > 0) {
323. totalBytesWritten += bytesRead;
324. }
325. printf("Data sent: %.2f%%\n", (double) totalBytesWritten / fileSize \* 100);
326. n++;
327. }
328. if (fclose(fp) == EOF) {
329. perror("appWrite - fclose");
330. return -1;
331. }
332. struct timeval endDataTime;
333. if (gettimeofday(&endDataTime, NULL) == -1) {
334. perror("appWrite - startup gettimeofday");
335. }
336. char endPacket[9 + strlen(filename)];
337. bzero(endPacket, 9 + strlen(filename));
338. endPacket[0] = END\_PACKET;
339. endPacket[1] = 0;
340. endPacket[2] = 4;
341. endPacket[3] = (fileSize & 0xFF000000) >> 24;
342. endPacket[4] = (fileSize & 0x00FF0000) >> 16;
343. endPacket[5] = (fileSize & 0x0000FF00) >> 8;
344. endPacket[6] = (fileSize & 0x000000FF);
345. endPacket[7] = 1;
346. endPacket[8] = strlen(filename);
347. memcpy(endPacket + 9, filename, strlen(filename));
348. if (llwrite(portFd, endPacket, 9 + strlen(filename)) == -1) {
349. printf("appWrite(): Failed to send end packet\n");
350. return -1;
351. }
352. if (llclose(portFd) == -1) {
353. printf("appWrite(): Failed to disconnect.\n");
354. return -1;
355. }
356. struct timeval endTime;
357. if (gettimeofday(&endTime, NULL) == -1) {
358. perror("appWrite - startup gettimeofday");
359. }
360. struct ExecTimes times = { &startTime, &startDataTime, &endDataTime,
361. &endTime };
362. printf("\n\n");
363. printTransmitterReport(totalBytesWritten, fileSize, &times);
364. return 0;
365. }

llAPI.h

1. #include <signal.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 <stdlib.h>
8. #include <unistd.h>
9. #include <strings.h>
10. #include <string.h>
11. #include <stdbool.h>
12. #define BAUDRATE B38400
13. //TODO - Ns and Nr.
14. #define FLAG 0x7E
15. #define F1 0x7D
16. #define F2 0x5E
17. #define A\_3 0x03
18. #define A\_1 0x01
19. #define C\_I 0x0
20. #define C\_SET 0x03
21. #define C\_UA 0x07
22. #define C\_RR 0x05
23. #define C\_REJ 0x01
24. #define C\_DISC 0xB
25. #define ESC 0x7D
26. // S and U frames
27. // A | C | BCC
28. #define A\_IND\_RESP 0
29. #define C\_IND\_RESP 1
30. #define BCC\_IND\_RESP 2
31. #define I\_FRAMES\_SEQ\_NUM\_BIT(x) (x >> 6)
32. #define S\_U\_FRAMES\_SEQ\_NUM\_BIT(x) (x >> 7)
33. #define TIMEOUT 3 //seconds
34. #define MAX\_TIME\_OUTS 5 // attempts
35. #define MAX\_REJS 5 //attempts
36. static bool timedOut = false;
37. static struct termios oldtio;
38. static enum CommsType global\_type;
39. void sigAlarmHandler(int sig) {
40. timedOut = true;
41. }
42. enum FrameTypeRes {
43. DATA, SET, DISC, UA, RR, REJ, IGNORE, ERROR
44. };
45. enum ReadFrameState {
46. AWAITING\_FLAG, AWAITING\_A, AWAITING\_C,
47. // C begin
48. FOUND\_I,
49. FOUND\_SET,
50. FOUND\_DISC,
51. FOUND\_UA,
52. FOUND\_RR,
53. FOUND\_REJ,
54. UNKNOWN\_C,
55. // C end
56. VALIDATED\_BCC\_I,
57. VALIDATED\_BCC\_OTHERS,
58. READING\_I\_DATA
59. };
60. enum ReadFrameState interpretC(char c) {
61. switch (c & 0x3F) { // ignore sequence number
62. case 0x0:
63. return FOUND\_I;
64. case 0x3:
65. return FOUND\_SET;
66. case 0xB:
67. return FOUND\_DISC;
68. case 0x7:
69. return FOUND\_UA;
70. case 0x5:
71. return FOUND\_RR;
72. case 0x1:
73. return FOUND\_REJ;
74. default:
75. return UNKNOWN\_C;
76. }
77. }
78. bool validBCC1(char A\_BYTE, char C, char BCC1) {
79. return BCC1 == (A\_BYTE ^ C);
80. }
81. enum FrameTypeRes readFrame(int fd, char \*\*frame, int \*frameLength) {
82. \*frame = malloc(5);
83. \*frameLength = 5;
84. (\*frame)[0] = FLAG;
85. enum ReadFrameState state = AWAITING\_FLAG;
86. enum FrameTypeRes frameTypeRes;
87. char buf;
88. int bytesRead;
89. while ((bytesRead = read(fd, &buf, 1)) != -1) {
90. if (bytesRead == 0) {
91. continue;
92. }
93. switch (state) {
94. case AWAITING\_FLAG:
95. if (buf == FLAG) {
96. state = AWAITING\_A;
97. }
98. break;
99. case AWAITING\_A:
100. if (buf != FLAG) {
101. state = AWAITING\_C;
102. (\*frame)[1] = buf;
103. }
104. break;
105. case AWAITING\_C:
106. state = interpretC(buf);
107. (\*frame)[2] = buf;
108. break;
109. case UNKNOWN\_C:
110. return IGNORE;
111. break;
112. case FOUND\_I:
113. state = VALIDATED\_BCC\_I;
114. (\*frame)[3] = buf;
115. if (!validBCC1((\*frame)[1], (\*frame)[2], (\*frame)[3])) {
116. return IGNORE;
117. }
118. break;
119. case FOUND\_SET:
120. state = VALIDATED\_BCC\_OTHERS;
121. frameTypeRes = SET;
122. (\*frame)[3] = buf;
123. if (!validBCC1((\*frame)[1], (\*frame)[2], (\*frame)[3])) {
124. return IGNORE;
125. }
126. break;
127. case FOUND\_DISC:
128. state = VALIDATED\_BCC\_OTHERS;
129. frameTypeRes = DISC;
130. (\*frame)[3] = buf;
131. if (!validBCC1((\*frame)[1], (\*frame)[2], (\*frame)[3])) {
132. return IGNORE;
133. }
134. break;
135. case FOUND\_UA:
136. state = VALIDATED\_BCC\_OTHERS;
137. frameTypeRes = UA;
138. (\*frame)[3] = buf;
139. if (!validBCC1((\*frame)[1], (\*frame)[2], (\*frame)[3])) {
140. return IGNORE;
141. }
142. break;
143. case FOUND\_RR:
144. state = VALIDATED\_BCC\_OTHERS;
145. frameTypeRes = RR;
146. (\*frame)[3] = buf;
147. if (!validBCC1((\*frame)[1], (\*frame)[2], (\*frame)[3])) {
148. return IGNORE;
149. }
150. break;
151. case FOUND\_REJ:
152. state = VALIDATED\_BCC\_OTHERS;
153. frameTypeRes = REJ;
154. (\*frame)[3] = buf;
155. if (!validBCC1((\*frame)[1], (\*frame)[2], (\*frame)[3])) {
156. return IGNORE;
157. }
158. break;
159. case VALIDATED\_BCC\_I:
160. (\*frame)[4] = buf;
161. if (buf == FLAG) {
162. return DATA;
163. } else {
164. state = READING\_I\_DATA;
165. }
166. break;
167. case VALIDATED\_BCC\_OTHERS:
168. if (buf != FLAG) {
169. return IGNORE;
170. } else {
171. (\*frame)[4] = FLAG;
172. return frameTypeRes;
173. }
174. case READING\_I\_DATA:
175. (\*frameLength)++;
176. \*frame = realloc(\*frame, \*frameLength);
177. (\*frame)[\*frameLength - 1] = buf;
178. if (buf == FLAG) {
179. return DATA;
180. }
181. }
182. }
183. return ERROR;
184. }
185. int setupConnection(char port[]) {
186. /\*\*
187. Open serial port device for reading and writing and not as controlling tty
188. because we don't want to get killed if linenoise sends CTRL-C.
189. \*/
190. int fd = open(port, O\_RDWR | O\_NOCTTY, S\_IRWXU | S\_IRWXG | S\_IRWXO);
191. if (fd < 0) {
192. perror(port);
193. exit(-1);
194. }
195. struct termios newtio;
196. if (tcgetattr(fd, &oldtio) == -1) { /\* save current port settings \*/
197. perror("tcgetattr");
198. exit(-1);
199. }
200. bzero(&newtio, sizeof(newtio));
201. newtio.c\_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;
202. newtio.c\_iflag = IGNPAR;
203. newtio.c\_oflag = 0;
204. /\* set input mode (non-canonical, no echo,...) \*/
205. newtio.c\_lflag = 0;
206. newtio.c\_cc[VTIME] = 1; /\* inter-character timer unused (em 100 ms)\*/
207. newtio.c\_cc[VMIN] = 0; /\* blocking read until 0 chars received \*/
208. tcflush(fd, TCIOFLUSH);
209. if (tcsetattr(fd, TCSANOW, &newtio) == -1) {
210. perror("tcsetattr");
211. exit(-1);
212. }
213. printf("New termios structure set\n");
214. return fd;
215. }
216. enum CommsType {
217. TRANSMITTER, RECEIVER
218. };
219. /\*\*
220. \* setMsg must already be allocated with 5 chars.
221. \*/
222. void makeSetMsg(char \*setMsg) {
223. setMsg[0] = FLAG;
224. setMsg[1] = A\_3;
225. setMsg[2] = C\_SET;
226. setMsg[3] = A\_3 ^ C\_SET;
227. setMsg[4] = FLAG;
228. }
229. /\*\*
230. \* uaMsg must already be allocated with 5 chars.
231. \*/
232. void makeUaMsg(char \*uaMsg) {
233. uaMsg[0] = FLAG;
234. uaMsg[1] = A\_3;
235. uaMsg[2] = C\_UA;
236. uaMsg[3] = A\_3 ^ C\_UA;
237. uaMsg[4] = FLAG;
238. }
239. int llopenTransmitter(int fd) {
240. int numTimeOuts = 0;
241. char setMsg[5];
242. int setMsgSize = 5;
243. makeSetMsg(setMsg);
244. do {
245. timedOut = false;
246. if (write(fd, setMsg, setMsgSize) == -1) {
247. perror("llopenTransmitter - write");
248. return -1;
249. }
250. alarm(3);
251. signal(SIGALRM, sigAlarmHandler);
252. enum FrameTypeRes res;
253. char \*frame = NULL;
254. int frameLength = 0;
255. do {
256. res = readFrame(fd, &frame, &frameLength);
257. free(frame);
258. } while (res != UA && !timedOut);
259. if (timedOut) {
260. numTimeOuts++;
261. if (numTimeOuts < MAX\_TIME\_OUTS) {
262. printf( "%d/%d: Timed out on connection establishment. Retrying.\n",
263. numTimeOuts, MAX\_TIME\_OUTS);
264. } else {
265. printf( "%d/%d: Timed out on connection establishment. Exiting.\n",
266. numTimeOuts, MAX\_TIME\_OUTS);
267. return -1;
268. }
269. }
270. } while (timedOut && numTimeOuts < MAX\_TIME\_OUTS);
271. return 0;
272. }
273. int llopenReceiver(int fd) {
274. char \*frame = NULL;
275. int frameLength = 0;
276. while (SET != readFrame(fd, &frame, &frameLength)) {
277. }
278. free(frame);
279. char uaMsg[5];
280. int uaMsgSize = 5;
281. makeUaMsg(uaMsg);
282. if (write(fd, uaMsg, uaMsgSize) == -1) {
283. perror("llopenReceiver - write");
284. return -1;
285. }
286. return 0;
287. }
288. int llopen(char port[], enum CommsType type) {
289. global\_type = type;
290. int fd = setupConnection(port);
291. switch (type) {
292. case TRANSMITTER:
293. if (llopenTransmitter(fd) == -1) {
294. #ifdef DEBUG
295. printf("llopen(): llopenTransmitter failed.\n");
296. #endif
297. return -1;
298. }
299. break;
300. case RECEIVER:
301. if (llopenReceiver(fd) == -1) {
302. #ifdef DEBUG
303. printf("llopen(): llopenReceiver failed.\n");
304. #endif
305. return -1;
306. }
307. break;
308. }
309. #ifdef DEBUG
310. printf("llopen success.\n");
311. #endif
312. return fd;
313. }
314. int makeFrame(char \*data, int dataLength, char seqNum, char \*\*frame,
315. int \*frameLength) {
316. \*frameLength = 4 + dataLength + 2;
317. \*frame = malloc(\*frameLength);
318. if (\*frame == NULL) {
319. perror("makeFrame - malloc");
320. return -1;
321. }
322. (\*frame)[0] = FLAG;
323. (\*frame)[1] = A\_3;
324. (\*frame)[2] = C\_I | (seqNum << 6);
325. (\*frame)[3] = (\*frame)[1] ^ (\*frame)[2];
326. char BCC2 = 0;
327. for (int dataInd = 0, frameInd = 4; dataInd < dataLength; dataInd++, frameInd++) {
328. (\*frame)[frameInd] = data[dataInd];
329. BCC2 ^= data[dataInd];
330. }
331. (\*frame)[\*frameLength - 2] = BCC2;
332. (\*frame)[\*frameLength - 1] = FLAG;
333. return 0;
334. }
335. /\*\*
336. \* Outputs stuffedFrame, allocating it. Its length is greater or equal to the original
337. \* frame's length.
338. \* Start and stop flags aren't stuffed.
339. \*/
340. int stuffFrame(char \*frame, int frameLength, char \*\*stuffedFrame,
341. int \*stuffedFrameLength) {
342. \*stuffedFrameLength = frameLength;
343. \*stuffedFrame = malloc(\*stuffedFrameLength);
344. if (\*stuffedFrame == NULL) {
345. printf("stuffFrame - malloc");
346. return -1;
347. }
348. (\*stuffedFrame)[0] = FLAG;
349. for (int unstuffedInd = 1, stuffedInd = 1; unstuffedInd < frameLength - 1;
350. unstuffedInd++, stuffedInd++) {
351. switch (frame[unstuffedInd]) {
352. case FLAG:
353. (\*stuffedFrameLength)++;
354. \*stuffedFrame = realloc(\*stuffedFrame, \*stuffedFrameLength);
355. if (\*stuffedFrame == NULL) {
356. perror("stuffFrame - realloc");
357. return -1;
358. }
359. (\*stuffedFrame)[stuffedInd++] = ESC;
360. (\*stuffedFrame)[stuffedInd] = FLAG ^ 0x20;
361. break;
362. case ESC:
363. (\*stuffedFrameLength)++;
364. \*stuffedFrame = realloc(\*stuffedFrame, \*stuffedFrameLength);
365. if (\*stuffedFrame == NULL) {
366. perror("stuffFrame - realloc");
367. return -1;
368. }
369. (\*stuffedFrame)[stuffedInd++] = ESC;
370. (\*stuffedFrame)[stuffedInd] = ESC ^ 0x20;
371. break;
372. default:
373. (\*stuffedFrame)[stuffedInd] = frame[unstuffedInd];
374. }
375. }
376. (\*stuffedFrame)[\*stuffedFrameLength - 1] = FLAG;
377. return 0;
378. }
379. int unstuffFrame(char \*stuffedFrame, int stuffedFrameLength, char \*\*frame,
380. int \*frameLength) {
381. \*frameLength = stuffedFrameLength;
382. \*frame = malloc(\*frameLength);
383. if (\*frame == NULL) {
384. perror("unstuffFrame - malloc");
385. return -1;
386. }
387. (\*frame)[0] = FLAG;
388. for (int stuffedInd = 1, unstuffedInd = 1; stuffedInd < stuffedFrameLength - 1;
389. stuffedInd++, unstuffedInd++) {
390. switch (stuffedFrame[stuffedInd]) {
391. case ESC:
392. (\*frameLength)--;
393. \*frame = realloc(\*frame, \*frameLength);
394. stuffedInd++;
395. (\*frame)[unstuffedInd] = stuffedFrame[stuffedInd] ^ 0x20;
396. break;
397. default:
398. (\*frame)[unstuffedInd] = stuffedFrame[stuffedInd];
399. break;
400. }
401. }
402. (\*frame)[\*frameLength - 1] = FLAG;
403. return 0;
404. }
405. int extractPacket(char \*\*packet, int \*packetLength, char \*frame, int frameLength) {
406. \*packetLength = -4 + frameLength - 2;
407. \*packet = malloc(\*packetLength);
408. if (\*packet == NULL) {
409. perror("extractPacket - malloc");
410. return -1;
411. }
412. memcpy(\*packet, frame + 4, \*packetLength);
413. return 0;
414. }
415. bool validPacketBCC(char \*packet, int packetLength, char BCC2) {
416. char acc = 0;
417. for (int i = 0; i < packetLength; i++) {
418. acc ^= packet[i];
419. }
420. return BCC2 == acc;
421. }
422. bool frameIsDuplicated(char \*frame, char previousSeqNum) {
423. if (previousSeqNum == -1) {
424. return false;
425. }
426. return previousSeqNum == (I\_FRAMES\_SEQ\_NUM\_BIT(frame[2]));
427. }
428. int sendReady(int fd, char seqNumber) {
429. int responseSize = 5;
430. char response[responseSize];
431. bzero(response, responseSize);
432. response[0] = FLAG;
433. response[1] = A\_3;
434. response[2] = C\_RR | (seqNumber << 7);
435. response[3] = A\_3 ^ (C\_RR | (seqNumber << 7));
436. response[4] = FLAG;
437. if (write(fd, response, responseSize) == -1) {
438. printf("sendReady(): write failed\n");
439. return -1;
440. }
441. return 0;
442. }
443. int sendRejection(int fd, char seqNumber) {
444. int responseSize = 5;
445. char response[responseSize];
446. bzero(response, responseSize);
447. response[0] = FLAG;
448. response[1] = A\_3;
449. response[2] = C\_REJ | (seqNumber << 7);
450. response[3] = A\_3 ^ (C\_REJ | (seqNumber << 7));
451. response[4] = FLAG;
452. if (write(fd, response, responseSize) == -1) {
453. printf("sendReady(): write failed\n");
454. return -1;
455. }
456. return 0;
457. }
458. /\*\*
459. \* @return Buffer length (bytes read), -1 if error.
460. \*/
461. int llread(int fd, char \*\*packet) {
462. static char previousSeqNum = -1;
463. bool rejected = false;
464. int numRejects = 0;
465. \*packet = NULL;
466. int packetLength = 0;
467. char frameC = 0;
468. bool discardedPacket = false;
469. do {
470. if (\*packet != NULL) {
471. free(\*packet);
472. \*packet = NULL;
473. }
474. discardedPacket = false;
475. rejected = false;
476. char \*stuffedFrame = NULL;
477. int stuffedFrameLength = 0;
478. while (true) {
479. enum FrameTypeRes res
480. = readFrame(fd, &stuffedFrame, &stuffedFrameLength);
481. if (res == DATA) {
482. break;
483. } else if (res == IGNORE) {
484. return 0;
485. }
486. free(stuffedFrame);
487. }
488. char \*frame = NULL;
489. int frameLength = 0;
490. if (unstuffFrame(stuffedFrame, stuffedFrameLength, &frame, &frameLength) == -1) {
491. #ifdef DEBUG
492. printf("llread(): unstuffFrame failed.\n");
493. #endif
494. free(stuffedFrame);
495. return -1;
496. }
497. free(stuffedFrame);
498. frameC = frame[2];
499. if (extractPacket(packet, &packetLength, frame, frameLength) == -1) {
500. #ifdef DEBUG
501. printf("llread(): extractPacket failed.\n");
502. #endif
503. return -1;
504. }
505. char BCC2 = frame[frameLength - 2];
506. if (!validPacketBCC(\*packet, packetLength, BCC2)) {
507. if (frameIsDuplicated(frame, previousSeqNum)) {
508. sendReady(fd, !I\_FRAMES\_SEQ\_NUM\_BIT(frameC));
509. rejected = false;
510. } else {
511. sendRejection(fd, !I\_FRAMES\_SEQ\_NUM\_BIT(frameC));
512. rejected = true;
513. }
514. } else {
515. if (frameIsDuplicated(frame, previousSeqNum)) {
516. free(\*packet);
517. \*packet = NULL;
518. discardedPacket = true;
519. }
520. sendReady(fd, !I\_FRAMES\_SEQ\_NUM\_BIT(frameC));
521. rejected = false;
522. }
523. free(frame);
525. if (rejected) {
526. numRejects++;
527. #ifdef DEBUG
528. printf("llread(): Packet rejected.\n");
529. #endif
530. }
531. } while (rejected && numRejects < MAX\_REJS);
532. previousSeqNum = I\_FRAMES\_SEQ\_NUM\_BIT(frameC);
533. if (!rejected && !discardedPacket) {
534. return packetLength;
535. } else {
536. return 0;
537. }
538. }
539. /\*\*
540. \* @return Bytes written, -1 if error.
541. \*/
542. int llwrite(int fd, char \*data, int dataLength) {
543. static char seqNum = 0;
544. char \*frame = NULL;
545. int frameLength = 0;
546. char \*stuffedFrame = NULL;
547. int stuffedFrameLength = 0;
548. char \*responseFrame = NULL;
549. int responseFrameLength = 0;
550. int numTimeOuts = 0;
551. int numRejects = 0;
552. bool accepted = false;
553. makeFrame(data, dataLength, seqNum, &frame, &frameLength);
554. stuffFrame(frame, frameLength, &stuffedFrame, &stuffedFrameLength);
555. free(frame);
556. do {
557. timedOut = false;
558. accepted = true;
559. if (write(fd, stuffedFrame, stuffedFrameLength) == -1) {
560. perror("llwrite - write");
561. free(stuffedFrame);
562. return -1;
563. }
564. alarm(3);
565. signal(SIGALRM, sigAlarmHandler);
566. enum FrameTypeRes res;
567. bool endRead = true;
568. do {
569. res = readFrame(fd, &responseFrame, &responseFrameLength);
570. endRead = true;
571. switch (res) {
572. case RR:
573. accepted = true;
574. seqNum = S\_U\_FRAMES\_SEQ\_NUM\_BIT(responseFrame[2]);
575. break;
576. case REJ:
577. accepted = false;
578. numRejects++;
579. break;
580. case IGNORE:
581. accepted = true;
582. break;
583. case ERROR:
584. timedOut = true;
585. break;
586. default:
587. endRead = false;
588. }
589. free(responseFrame);
590. } while (!timedOut && !endRead);
591. alarm(0);
592. if (timedOut) {
593. ++numTimeOuts;
594. if (numTimeOuts < MAX\_TIME\_OUTS) {
595. printf("%d/%d: Timed out while sending packet. Retrying.\n",
596. numTimeOuts, MAX\_TIME\_OUTS);
597. } else {
598. printf("%d/%d: Timed out while sending packet. Exiting.\n",
599. numTimeOuts, MAX\_TIME\_OUTS);
600. alarm(0);
601. free(stuffedFrame);
602. return -1;
603. }
604. }
605. } while ((timedOut && numTimeOuts < MAX\_TIME\_OUTS)
606. || (!accepted && numRejects < MAX\_REJS));
607. free(stuffedFrame);
608. if (numTimeOuts >= MAX\_TIME\_OUTS) {
609. return 0;
610. } else {
611. return dataLength;
612. }
613. }
614. /\*\*
615. \* discMsg must already be allocated with 5 chars.
616. \*/
617. void makeTransDiscMsg(char \*discMsg) {
618. discMsg[0] = FLAG;
619. discMsg[1] = A\_3;
620. discMsg[2] = C\_DISC;
621. discMsg[3] = A\_3 ^ C\_DISC;
622. discMsg[4] = FLAG;
623. }
624. /\*\*
625. \* discMsg must already be allocated with 5 chars.
626. \*/
627. void makeReceiverDiscMsg(char \*discMsg) {
628. discMsg[0] = FLAG;
629. discMsg[1] = A\_1;
630. discMsg[2] = C\_DISC;
631. discMsg[3] = A\_1 ^ C\_DISC;
632. discMsg[4] = FLAG;
633. }
634. void makeTransUaMsg(char \*uaMsg) {
635. uaMsg[0] = FLAG;
636. uaMsg[1] = A\_1;
637. uaMsg[2] = C\_UA;
638. uaMsg[3] = A\_1 ^ C\_UA;
639. uaMsg[4] = FLAG;
640. }
641. /\*\*
642. \* Returns 1 if success, -1 if error.
643. \*/
644. int llcloseTransmitter(int fd) {
645. int numTimeOuts = 0;
646. int discMsgSize = 5;
647. char discMsg[5];
648. makeTransDiscMsg(discMsg);
649. do {
650. timedOut = false;
651. if (write(fd, discMsg, discMsgSize) == -1) {
652. perror("llcloseTransmitter - write");
653. return -1;
654. }
655. alarm(3);
656. signal(SIGALRM, sigAlarmHandler);
657. enum FrameTypeRes res;
658. char \*frame = NULL;
659. int frameLength = 0;
660. do {
661. res = readFrame(fd, &frame, &frameLength);
662. free(frame);
663. } while (res != DISC && !timedOut);
664. alarm(0);
665. if (timedOut) {
666. numTimeOuts++;
667. if (numTimeOuts < MAX\_TIME\_OUTS) {
668. printf("%d/%d: Timed out on disconnection. Retrying.\n",
669. numTimeOuts, MAX\_TIME\_OUTS);
670. } else {
671. printf("%d/%d: Timed out on disconnection. Exiting.\n",
672. numTimeOuts, MAX\_TIME\_OUTS);
673. return -1;
674. }
675. }
676. } while (timedOut && numTimeOuts < MAX\_TIME\_OUTS);
677. int uaMsgSize = 5;
678. char uaMsg[5];
679. makeTransUaMsg(uaMsg);
680. if (write(fd, uaMsg, uaMsgSize) == -1) {
681. perror("llcloseTransmitter - write");
682. return -1;
683. }
684. return 1;
685. }
686. /\*\*
687. \* Returns 1 if success, -1 if error.
688. \*/
689. int llcloseReceiver(int fd) {
690. enum FrameTypeRes res;
691. char \*frame = NULL;
692. int frameLength = 0;
693. do {
694. res = readFrame(fd, &frame, &frameLength);
695. free(frame);
696. } while (res != DISC);
697. int discMsgSize = 5;
698. char discMsg[5];
699. makeReceiverDiscMsg(discMsg);
701. int numTimeOuts = 0;
702. do {
703. timedOut = false;
704. if (write(fd, discMsg, discMsgSize) == -1) {
705. perror("llcloseReceiver - write");
706. return -1;
707. }
708. alarm(3);
709. signal(SIGALRM, sigAlarmHandler);
710. enum FrameTypeRes res;
711. char \*frame = NULL;
712. int frameLength = 0;
713. do {
714. res = readFrame(fd, &frame, &frameLength);
715. free(frame);
716. } while (res != UA && !timedOut);
717. alarm(0);
719. if (timedOut) {
720. numTimeOuts++;
721. if (numTimeOuts < MAX\_TIME\_OUTS) {
722. printf( "%d/%d: Timed out on disconnection acknowledgement. Retrying.\n",
723. numTimeOuts, MAX\_TIME\_OUTS);
724. } else {
725. printf( "%d/%d: Timed out on disconnection acknowledgement. Exiting.\n",
726. numTimeOuts, MAX\_TIME\_OUTS);
727. return -1;
728. }
729. }
730. } while (timedOut && numTimeOuts < MAX\_TIME\_OUTS);
731. return 1;
732. }
733. /\*\*
734. \* Returns 1 if success, -1 if error.
735. \*/
736. int llclose(int fd) {
737. switch (global\_type) {
738. case TRANSMITTER:
739. return llcloseTransmitter(fd);
740. case RECEIVER:
741. return llcloseReceiver(fd);
742. default:
743. #ifdef DEBUG
744. printf("llclose(): Invalid CommsType.\n");
745. #endif
746. return -1;
747. }
748. }

llReadTest.c

1. #include "llAPI.h"
2. int main() {
3. char port[] = "/dev/ttyS0";
4. int fd = llopen(port, RECEIVER);
5. if (fd == -1) {
6. printf("llopen failed.\n");
7. }
8. char \*msg = NULL;
9. int msgSize = llread(fd, &msg);
10. printf("Data size: %d\n", msgSize);
11. printf("HEX:\n");
12. for (int i = 0; i < msgSize; i++) {
13. printf("%x ", msg[i]);
14. }
15. printf("\n");
16. printf("Chars:\n");
17. for (int i = 0; i < msgSize; i++) {
18. printf("%c ", msg[i]);
19. }
20. int res = llclose(fd);
21. printf("llclose res = %d\n", res);
22. }

llWriteTest.c

1. #include "llAPI.h"
2. #define FLAG 0x7E
3. #define ESC 0x7D
4. int main() {
5. char port[] = "/dev/ttyS0";
6. int msgSize = 12;
7. char msg[12] = { FLAG, ESC, 'a', 0x5, '0', ESC, 0, FLAG, 5, ESC, ESC, FLAG };
8. // char msg[12] = {ESC, 'a', 0x5, '0', 0, 5};
9. int fd = llopen(port, TRANSMITTER);
10. int bytesWritten = llwrite(fd, msg, msgSize);
11. printf("bytesWritten = %d\n", bytesWritten);
12. int res = llclose(fd);
13. printf("llclose res = %d\n", res);
14. }

noncanonical.c

1. /\*Non-Canonical Input Processing\*/
2. #include "appAPI.h"
3. #define \_POSIX\_SOURCE 1 /\* POSIX compliant source \*/
4. #define FALSE 0
5. #define TRUE 1
6. volatile int STOP = FALSE;
7. int main(int argc, char\*\* argv) {
8. #ifdef DEBUG
9. printf("Debug mode: ON.\n");
10. #endif
11. if (argc < 2) {
12. printf("Usage:\tSerialPort\n\tex: /dev/ttyS0\n");
13. exit(1);
14. }
15. if (appRead(argv[1]) == -1) {
16. printf("appRead() failed.\n");
17. return -1;
18. }
19. return 0;
20. }

writenoncanonical.c

1. /\*Non-Canonical Input Processing\*/
2. #include "appAPI.h"
3. #define MODEMDEVICE "/dev/ttyS1"
4. #define \_POSIX\_SOURCE 1 /\* POSIX compliant source \*/
5. #define FALSE 0
6. #define TRUE 1
7. volatile int STOP = FALSE;
8. int main(int argc, char\*\* argv) {
9. if (argc < 2) {
10. printf("Usage:\tSerialPort FileName\n");
11. printf("\tex: /dev/ttyS0 image.png\n");
12. exit(1);
13. }
14. if (appWrite(argv[1], argv[2]) == -1) {
15. printf("appWrite() failed.\n");
16. return -1;
17. }
18. return 0;
19. }