



FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

3ºano – MIEIC – Novembro 2017

Protocolo de Ligação de Dados

RCOM

Turma 2

João Francisco Veríssimo Dias Esteves – up201505145

João Miguel Matos Monteiro – up201506130

Maria Eduarda Santos Cunha – up201506524



Índice

1. Sumário	3
2. Introdução	3
3. Arquitetura e Estrutura do Código	3
3.1. Camada de Ligação de Dados.....	4
3.2. Camada de Aplicação	4
3.3. Interface.....	4
4. Casos de Uso Principais.....	5
5. Protocolo de Ligação Lógica	6
5.1. llopen() e llclose()	7
5.2. llwrite() e llread().....	7
6. Protocolo de Aplicação	8
6.1. appWrite().....	8
6.2. appRead()	8
7. Validação	9
8. Eficiência do Protocolo de Ligação de Dados	9
9. Conclusão	10
10. Anexos.....	11



1. 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.

2. 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 perspectiva 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 e Estrutura do Código: Blocos funcionais, 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;

Eficiência do Protocolo de Ligação Lógica: Medidas de tempos de transferência com a variação de alguns fatores.

3. 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.



3.1. 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.

3.2. 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 *structs timeval* para os tempos de início do programa, início da transferência, fim da transferência e fim do programa.

3.3. 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.

Durante o decorrer do programa, a interface apresenta estatísticas relativas ao envio do ficheiro, como x de y bytes enviados e o tempo de envio. Ou, ainda, mensagens de alerta para indicar, por exemplo, a ocorrência de timeouts.

4. Casos de Uso Principais

- Escolha do ficheiro a enviar;
- Configuração da ligação;
- Estabelecimento da ligação;
- Envio dos dados do ficheiro pelo emissor;
- Receção dos dados pelo recetor e respetiva escrita no ficheiro de output;
- Impressão de dados na consola relativos a progresso do envio e tempo ou de erros no processo, adaptados ao emissor e recetor;
- Término da ligação

Da perspetiva do transmissor (figura 1), primeiro é chamada a função *appWrite()* com os argumentos porta série e nome do ficheiro a transmitir. Dentro desta função, começa-se com o *llopen()* de forma a abrir a ligação, de seguida um *llwrite()* com o pacote de controlo de início, um ciclo de *llwrite()*'s para cada pacote constituído por x bytes do ficheiro total, *llwrite()* com o pacote de controlo de fim e, finalmente, recorremos a *llclose()* para fechar a ligação. No final, a função *printTransmitterReport()* procede à impressão de um relatório com informação relativa ao número de bytes transmitidos, o tempo que leva a transmissão do ficheiro e o tempo total de todo o processo.

Da perspetiva do recetor (figura 2), primeiramente é chamada a função *appRead()* com o argumento porta série. Todas as chamadas de funções são semelhantes às do transmissor, exceto em vez de *llwrite()*'s, recorremos a *llread()*'s e surge a chamada às funções *processStartPacket()*, *processDataPacket()* e *processEndPacket()*, que consistem em, respetivamente, saber o tamanho e nome do ficheiro a enviar, obter cada fragmento do ficheiro



e verificar se o pacote de controlo de fim é igual ao de início. Ainda, a função *writeLocalFile()* é utilizada para escrever para o ficheiro de destino a informação recebida do transmissor.

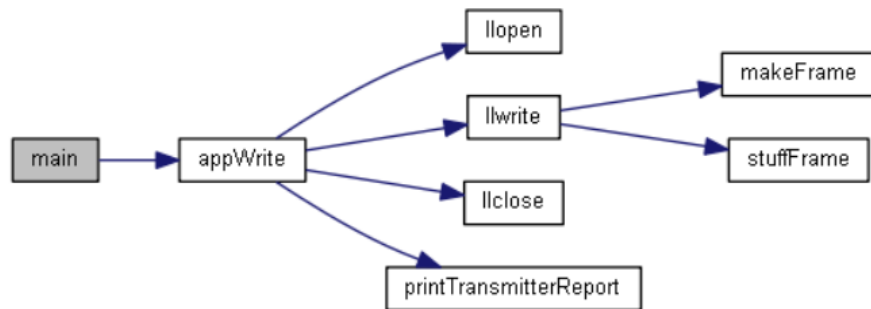


Figure 2: Transmissor

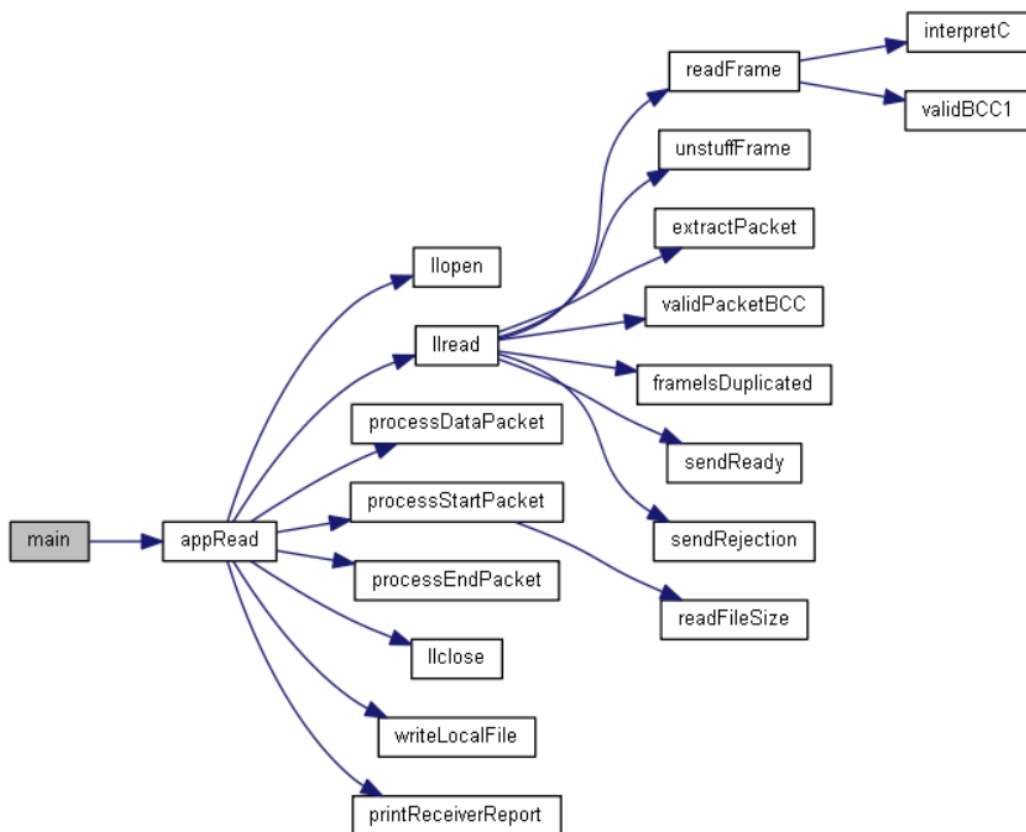


Figure 2: Recetor

5. Protocolo de Ligação Lógica

O Protocolo de Ligação Lógica instaura a comunicação de dados fiável entre 2 sistemas ligados por uma porta série. Encontra-se implementado na *llAPI.h*.

Os principais aspetos funcionais deste protocolo são a configuração da porta série para o



seu uso durante a transmissão, o estabelecimento da ligação, o envio e recepção de dados e, por fim, o término da ligação.

5.1. `llopen()` e `llclose()`

A função `llopen()` tem a responsabilidade de estabelecer a ligação, retornando o descritor de ficheiro da porta série fornecida como argumento. Em primeiro lugar, a porta série é configurada para que, entre outros, esteja em modo não-canónico e a leitura desta não bloqueie se não houverem caracteres para ler. Em seguida, o transmissor envia uma trama SET e espera por uma trama UA e o recetor espera por um SET e envia um UA. A espera pelo transmissor do UA após o envio do SET é protegida por um *time out*, sendo que, se após um certo tempo do envio ainda não foi recebido um UA e ainda não se atingiu o número máximo de *timeouts*, o SET é reenviado.

A função `llclose()` serve para terminar a ligação. O transmissor envia uma trama DISC, aguarda uma trama DISC e envia uma trama UA, enquanto o recetor espera por um DISC, envia um DISC e espera por um UA. A espera pelos DISC e UA em ambos transmissor e recetor também está protegida por *timeout*.

5.2. `llwrite()` e `llread()`

A função `llwrite()`, usada pelo transmissor, tem a responsabilidade de enviar dados com êxito. Os dados são encapsulados numa trama I, contendo 1 byte FLAG no início e no fim. Para evitar que os bytes da secção de dados sejam interpretados como FLAGS, é aplicado o processo de *byte stuffing*, substituindo cada uma destas falsas FLAGS por um byte de escape, ESC e pelo OU-exclusivo da FLAG com 0x20 e substituindo cada falso ESC por um ESC e pelo OU-exclusivo do ESC com 0x20. Esta trama é, então, enviada, esperando-se uma trama RR ou REJ como resposta, sendo este envio protegido por *timeout*. Caso a resposta seja um REJ e ainda não se tenha atingido o número máximo de tramas rejeitadas, procede-se ao reenvio da trama I.

A função `llread()`, usada pelo recetor, tem a responsabilidade de receber dados com êxito e reportar a sua recepção ao transmissor. Os dados são recebidos dentro de uma trama I, na qual é necessário, em primeiro lugar, realizar-se o processo de *byte unstuffing*, que consiste no inverso de *byte stuffing*: converter cada conjunto de 2 bytes começado por ESC em 1 byte que é o OU-exclusivo do byte originalmente a seguir ao ESC com 0x20. A secção de dados



(incluindo o respetivo BCC) é de seguida extraído. É enviado um RR caso o BCC dos dados seja válido, embora o pacote não seja lido caso a trama for um duplicado, e também se o BCC dos dados for inválido, mas a trama for um duplicado. É enviado um REJ caso o BCC dos dados seja inválido e a trama não for um duplicado. Se for o REJ a ser enviado e o número máximo de rejeitados ainda não tenha sido atingido, é esperado que seja lida outra trama I.

6. Protocolo de Aplicação

O Protocolo de Aplicação está implementado na *appAPI.h* e depende da camada de ligação de dados, *llAPI.h*.

Os principais aspetos funcionais deste protocolo são a leitura do ficheiro a transmitir, a escrita do ficheiro recebido e o envio e receção do ficheiro.

6.1. *appWrite()*

Esta é a função chamada pelo transmissor, que carrega em memória o ficheiro a transmitir e o envia pela porta série indicada e apresenta um relatório final. Primeiro, é estabelecida a ligação através da função *llopen()*. A seguir, sempre recorrendo à função *llwrite()*, é enviado um pacote de controlo de início com o tamanho e o nome do ficheiro, depois, à medida que é lido o ficheiro, este é enviado pacote a pacote e, finalmente, é enviado um pacote de controlo de fim, igual ao de início, excetuando o primeiro byte, que identifica o pacote como sendo de controlo de início, de dados ou de controlo de fim. De seguida, é terminada a ligação pela função *llclose()* e, por fim, é apresentado um relatório final com os bytes transmitidos e os totais do ficheiro e os tempos de execução total e da transferência.

6.2. *appRead()*

Esta é a função chamada pelo recetor, que recebe um ficheiro pela porta série indicada e o escreve localmente e apresenta um relatório final. Para começar, é estabelecida a ligação através da função *llopen()*. Seguidamente, são lidos os pacotes sucessivos do ficheiro através da função *llread()*, classificando em pacotes de controlo de início, de controlo de fim e de dados pelo primeiro byte do pacote lido, acabando a leitura assim que for lido o pacote de controlo de fim. Os pacotes de controlo de início e de fim deverão ser iguais, exceto o primeiro byte, e deverão ter obrigatoriamente o tamanho do ficheiro e opcionalmente o nome do ficheiro. Depois é terminada a ligação pela função *llclose()* e, finalmente, o ficheiro recebido é



escrito localmente pela função `writeLocalFile()`. É apresentado um relatório final com os bytes recebidos dos totais indicados pelos pacotes de controlo, o número de incorrespondências nos números sequenciais dos pacotes e os tempos de execução total e da transferência.

7. Validação

De forma a testar a integralidade do protocolo implementado, procedemos à realização de vários testes, nomeadamente a transferência do ficheiro sem qualquer tipo de obstáculo, a transferência fechando e voltando a abrir a porta de série, fechando-a até ao timeout e com a introdução de erros.

Para os diferentes tipos de teste utilizou-se vários ficheiros: `pinguim.gif` e alguns outros, de entre os quais ficheiros de texto e imagens com tamanhos mais pesados.

Como resposta a estes testes, é imprimido na consola o progresso em percentagem do envio do ficheiro, os timeouts, os *rejects* e quantos bytes foram enviados dos que eram supostos.

8. Eficiência do Protocolo de Ligação de Dados

Foram medidas as médias dos tempos de transferência do `pinguim.gif`, de 10.7KB, para diferentes tamanhos de pacotes de dados das tramas I e diferentes Baudrates, apresentados na seguinte tabela (tamanho em Bytes, tempos em segundos).

Baudrate/ Tamanho	256	1024	2048
B9600	13.00	12.24	12.40
B38400	3.10	3.00	3.00
B57600	2.06	2.04	2.06
B230400	2.06	2.04	2.06

Concluimos que, normalmente, o tamanho dos pacotes não afeta significativamente o tempo de transferência, embora o Baudrate seja um grande fator até certo valor, a partir do qual é irrelevante. No entanto, se o Baudrate for demasiado pequeno para o tamanho do pacote, este demora demais a ser enviado e pode resultar num timeout, impossibilitando a transferência de dados.



Neste protocolo é usado o método *Stop&Wait*, que consiste em o transmissor esperar por uma resposta sempre que envia uma trama. O recetor enviará diferentes tramas conforme as recebidas, enviando um UA caso receba um SET devidamente no início da conexão, um RR ou REJ por cada trama de dados recebida, e um DISC se receber um DISC no final. Quanto aos RR e REJ, o RR será enviado se a trama é recebida sem erros e o REJ caso tenha erros. Deste modo, é garantido que não hajam perdas de pacotes pois estes serão reenviados caso não haja uma resposta ou esta assinala um erro.

9. 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.



10. Anexos

appAPI.h

```

1  #include "IIAPI.h"
2  #include <stdio.h>
3  #include <math.h>
4  #include <sys/time.h>
5
6  #define C_APP 0
7  #define TLV_T 0
8  #define TLV_L 1
9  #define TLV_V 2
10
11 #define DATA_N 1
12 #define DATA_L2 2
13 #define DATA_L1 3
14 #define DATA_P1 4
15
16 #define T_FILE_SIZE 0
17 #define T_FILE_NAME 1
18
19 #define DATA_PACKET 1
20 #define START_PACKET 2
21 #define END_PACKET 3
22
23 struct ExecTimes {
24     struct timeval *startTime;
25     struct timeval *startDataTime;
26     struct timeval *endDataTime;
27     struct timeval *endTime;
28 };
29
30 void readFileSize(char *fileSizeChars, int *fileLength, int arrayLength) {
31     *fileLength = 0;
32     int hexOrder = 0;
33     for (int i = arrayLength - 1; i >= 0; i--) {
34         *fileLength += (fileSizeChars[i] & 0xF) * pow(16, hexOrder++);
35         *fileLength += ((fileSizeChars[i] & 0xF0) >> 4) * pow(16, hexOrder);
36         hexOrder++;
37     }
38 }
39
40 /**
41  Reads data in packet to fileBuffer sequentially, reallocating it.
42  */
43 int processDataPacket(char *packet, char **fileBuffer, int *fileBufferLength,
44 int *seqNumMismatches) {
45     static int prevSeqNum = -1;
46     unsigned char sequenceNumber = packet[DATA_N];
47     if (prevSeqNum == -1) {
48         prevSeqNum = sequenceNumber;
49     } else if ((prevSeqNum + 1) % 255 != sequenceNumber) {
50         printf("Warning: Sequence number mismatch in data packet.\n");
51         (*seqNumMismatches)++;
52     }
53     #ifdef DEBUG
54     printf("prevSeqNum = %d | sequenceNumber = %d\n",
55         prevSeqNum, sequenceNumber);
56     #endif
57 }

```



```
55     #endif
56     }
57
58     int dataSize = 256 * (unsigned char) packet[DATA_L2]
59     +(unsigned char) packet[DATA_L1];
60     *fileBuffer = realloc(*fileBuffer, *fileBufferLength + dataSize);
61     for (int i = 0; i < dataSize; i++) {
62         (*fileBuffer)[*fileBufferLength + i] = packet[DATA_P1 + i];
63     }
64     *fileBufferLength += dataSize;
65
66     prevSeqNum = sequenceNumber;
67     return 0;
68 }
69
70 /**
71 Reads file length and filename from packet, if they exist.
72 */
73 int processStartPacket(char *packet, int packetLength, int *fileLength, char **filename)
74 {
75     bool setName = false, setSize = false;
76     int bytesRead = 1;
77
78     while (bytesRead < packetLength) {
79         int vLength = packet[bytesRead + TLV_L];
80         switch (packet[bytesRead + TLV_T]) {
81             case T_FILE_SIZE:
82                 if (setSize) {
83                     break;
84                 }
85                 char *fileSizeChars = malloc(vLength + 1);
86                 memcpy(fileSizeChars, packet + bytesRead + TLV_V, vLength);
87                 fileSizeChars[vLength] = '\0';
88                 readFileSize(fileSizeChars, fileLength, vLength);
89                 free(fileSizeChars);
90                 setSize = true;
91                 break;
92             case T_FILE_NAME:
93                 if (setName) {
94                     break;
95                 }
96                 *filename = malloc(vLength);
97                 if (filename == NULL) {
98                     perror("processStartPacket - malloc");
99                     return -1;
100                 }
101                 memcpy(*filename, packet + bytesRead + TLV_V, vLength);
102                 setName = true;
103                 break;
104         }
105         bytesRead += 2 + vLength;
106     }
107
108     if (!setSize) {
109         printf("processControlPacket(): Start packet did not contain file size.\n");
110         return -1;
111     }
112 }
```



```
113     return 0;
114 }
115
116 int processEndPacket(char *endPacket, char *startPacket, int packetLength) {
117     for (int i = 1; i < packetLength; i++) {
118         if (endPacket[i] != startPacket[i]) {
119             printf("processEndPacket(): End packet does not match start packet.\n");
120             return -1;
121         }
122     }
123     return 0;
124 }
125
126 int writeLocalFile(char *filename, char *fileBuffer, int fileBufferLength) {
127     FILE *fp = fopen(filename, "wb");
128
129     if (fp == NULL) {
130         perror("writeLocalFile - fopen");
131         return -1;
132     }
133
134     if (fwrite(fileBuffer, 1, fileBufferLength, fp) == -1) {
135         perror("writeLocalFile - write");
136         fclose(fp);
137         return -1;
138     }
139
140     if (fclose(fp) == EOF) {
141         perror("writeLocalFile - fclose");
142         return -1;
143     }
144     return 0;
145 }
146
147 void printReceiverReport(int receivedBytes, int originalFileSize,
148 int seqNumMismatches, struct ExecTimes *times) {
149     time_t totalSeconds = times->endTime->tv_sec - times->startTime->tv_sec;
150     suseconds_t totalMicroseconds;
151     if (times->endTime->tv_usec > times->startTime->tv_usec) {
152         totalMicroseconds = times->endTime->tv_usec - times->startTime->tv_usec;
153     } else {
154         totalMicroseconds = times->startTime->tv_usec - times->endTime->tv_usec;
155     }
156
157     time_t dataSeconds = times->endTime->tv_sec
158 - times->startTime->tv_sec;
159     suseconds_t dataMicroseconds;
160     if (times->endTime->tv_usec > times->startTime->tv_usec) {
161         dataMicroseconds = times->endTime->tv_usec
162 - times->startTime->tv_usec;
163     } else {
164         dataMicroseconds = times->startTime->tv_usec
165 - times->endTime->tv_usec;
166     }
167
168     printf("Received bytes: %d out of %d.\n", receivedBytes, originalFileSize);
169     printf("Sequence number mismatches: %d.\n", seqNumMismatches);
170     printf("Data transfer time: %.3fs.\n", dataSeconds + (double) dataMicroseconds
```



```
171     / pow(10, 6));
172     printf("Total time: %.3fs.\n", totalSeconds + (double) totalMicroseconds
173     / pow(10, 6));
174 }
175
176 int appRead(char port[]) {
177     struct timeval startTime;
178     if (gettimeofday(&startTime, NULL) == -1) {
179         perror("appWrite - startup gettimeofday");
180     }
181     char *filename = NULL;
182     char *fileBuffer = NULL, *packet = NULL, *startPacket = NULL;
183     int fileBufferLength = 0;
184     int fileLength = 0;
185     int packetLength = 0;
186     bool finished = false;
187     int seqNumMismatches = 0;
188     int fd = llopen(port, RECEIVER);
189
190     struct timeval startDataTime;
191     if (gettimeofday(&startDataTime, NULL) == -1) {
192         perror("appWrite - startup gettimeofday");
193     }
194
195     while (!finished) {
196         if ((packetLength = lread(fd, &packet)) == -1) {
197             printf("appRead(): lread() failed\n");
198             return -1;
199         }
200         if (packet == NULL) {
201             continue;
202         }
203         switch (packet[C_APP]) {
204             case DATA_PACKET:
205                 processDataPacket(packet, &fileBuffer, &fileBufferLength,
206                 &seqNumMismatches);
207                 printf("Data received: %.2f%%\n", (double) fileBufferLength / fileLength * 100);
208                 break;
209             case START_PACKET:
210                 if (processStartPacket(packet, packetLength, &fileLength, &filename)
211                 == -1) {
212                     free(packet);
213                     return -1;
214                 }
215                 startPacket = malloc(packetLength);
216                 memcpy(startPacket, packet, packetLength);
217                 break;
218             case END_PACKET:
219                 if (processEndPacket(packet, startPacket, packetLength) == -1) {
220                     printf("appRead(): processEndPacket failed.\n");
221                     free(packet);
222                     free(fileBuffer);
223                     return -1;
224                 }
225                 finished = true;
226                 break;
227         }
228         free(packet);
```



```
229     }
230
231     free(startPacket);
232
233     struct timeval endTime;
234     if (gettimeofday(&endTime, NULL) == -1) {
235         perror("appWrite - startup gettimeofday");
236     }
237
238     if (fclose(fd) == -1) {
239         printf("appRead(): fclose() failed\n");
240         free(filename);
241         free(fileBuffer);
242         return -1;
243     }
244
245     if (writeLocalFile(filename, fileBuffer, fileBufferLength) == -1) {
246         printf("appRead(): writeLocalFile() failed.\n");
247         free(filename);
248         free(fileBuffer);
249         return -1;
250     }
251
252     free(filename);
253     free(fileBuffer);
254
255     struct timeval endTime;
256     if (gettimeofday(&endTime, NULL) == -1) {
257         perror("appWrite - startup gettimeofday");
258     }
259
260     struct ExecTimes times = { &startTime, &startDataTime, &endTime,
261                               &endTime };
262
263     printf("\n\n");
264     printReceiverReport(fileBufferLength, fileLength, seqNumMismatches, &times);
265
266     return 0;
267 }
268
269 void printTransmitterReport(int bytesSent, int fileSize,
270                             struct ExecTimes *times) {
271     time_t totalSeconds = times->endTime->tv_sec - times->startTime->tv_sec;
272     suseconds_t totalMicroseconds;
273     if (times->endTime->tv_usec > times->startTime->tv_usec) {
274         totalMicroseconds = times->endTime->tv_usec - times->startTime->tv_usec;
275     } else {
276         totalMicroseconds = times->startTime->tv_usec - times->endTime->tv_usec;
277     }
278
279     time_t dataSeconds = times->endTime->tv_sec
280     - times->startDataTime->tv_sec;
281     suseconds_t dataMicroseconds;
282     if (times->endTime->tv_usec > times->startDataTime->tv_usec) {
283         dataMicroseconds = times->endTime->tv_usec
284         - times->startDataTime->tv_usec;
285     } else {
286         dataMicroseconds = times->startDataTime->tv_usec
```



```
287     - times->endTime->tv_usec;
288 }
289
290 printf("Transmitted bytes: %d out of %d.\n", bytesSent, fileSize);
291 printf("Data transfer time: %.3fs.\n",
292     dataSeconds + (double) dataMicroseconds / pow(10, 6));
293 printf("Total time: %.3fs.\n",
294     totalSeconds + (double) totalMicroseconds / pow(10, 6));
295 }
296
297 int appWrite(char port[], char filename[]) {
298     struct timeval startTime;
299     if (gettimeofday(&startTime, NULL) == -1) {
300         perror("appWrite - startup gettimeofday");
301     }
302     int portFd = llopen(port, TRANSMITTER);
303     if (portFd == -1) {
304         printf("appWrite(): Failed to open connection.\n");
305         return -1;
306     }
307
308     int fileSize = -1;
309     FILE *fp = fopen(filename, "rb");
310     if (fp == NULL) {
311         perror("appWrite - fopen");
312         return -1;
313     }
314
315     struct stat statBuf;
316     stat(filename, &statBuf);
317     fileSize = statBuf.st_size;
318
319     char startPacket[9 + strlen(filename)];
320     bzero(startPacket, 9 + strlen(filename));
321     startPacket[0] = START_PACKET;
322     startPacket[1] = 0;
323     startPacket[2] = 4;
324     startPacket[3] = (fileSize & 0xFF000000) >> 24;
325     startPacket[4] = (fileSize & 0x00FF0000) >> 16;
326     startPacket[5] = (fileSize & 0x0000FF00) >> 8;
327     startPacket[6] = (fileSize & 0x000000FF);
328     startPacket[7] = 1;
329     startPacket[8] = strlen(filename);
330     memcpy(startPacket + 9, filename, strlen(filename));
331
332     if (llwrite(portFd, startPacket, 9 + strlen(filename)) == -1) {
333 #ifdef DEBUG
334         printf("appWrite(): Failed to send start packet.\n");
335 #endif
336         fclose(fp);
337         return -1;
338     }
339
340     struct timeval startDataTime;
341     if (gettimeofday(&startDataTime, NULL) == -1) {
342         perror("appWrite - startup gettimeofday");
343     }
344 }
```




```
345     char buffer[1024];
346     char n = 0;
347     int bytesRead = -2;
348     int totalBytesWritten = 0;
349     while (bytesRead = fread(buffer, 1, 1024, fp)) {
350         char dataPacket[bytesRead + 4];
351         bzero(dataPacket, bytesRead + 4);
352
353         dataPacket[0] = DATA_PACKET;
354         dataPacket[1] = n % 255;
355         dataPacket[2] = bytesRead / 256;
356         dataPacket[3] = bytesRead % 256;
357
358         memcpy(dataPacket + 4, buffer, bytesRead);
359
360         int res = -2;
361         if ((res = llwrite(portFd, dataPacket, bytesRead + 4)) == -1) {
362             #ifdef DEBUG
363                 printf("appWrite(): Failed to send data packet.\n");
364             #endif
365             fclose(fp);
366             return -1;
367         }
368         if (res > 0) {
369             totalBytesWritten += bytesRead;
370         }
371         printf("Data sent: %.2f%%\n", (double) totalBytesWritten / fileSize * 100);
372         n++;
373     }
374
375     if (fclose(fp) == EOF) {
376         perror("appWrite - fclose");
377         return -1;
378     }
379
380     struct timeval endDataTime;
381     if (gettimeofday(&endDataTime, NULL) == -1) {
382         perror("appWrite - startup gettimeofday");
383     }
384
385     char endPacket[9 + strlen(filename)];
386     bzero(endPacket, 9 + strlen(filename));
387     endPacket[0] = END_PACKET;
388     endPacket[1] = 0;
389     endPacket[2] = 4;
390     endPacket[3] = (fileSize & 0xFF000000) >> 24;
391     endPacket[4] = (fileSize & 0x00FF0000) >> 16;
392     endPacket[5] = (fileSize & 0x0000FF00) >> 8;
393     endPacket[6] = (fileSize & 0x000000FF);
394     endPacket[7] = 1;
395     endPacket[8] = strlen(filename);
396     memcpy(endPacket + 9, filename, strlen(filename));
397
398     if (llwrite(portFd, endPacket, 9 + strlen(filename)) == -1) {
399         printf("appWrite(): Failed to send end packet\n");
400         return -1;
401     }
402 }
```



```
403     if (!fclose(portFd) == -1) {
404         printf("appWrite(): Failed to disconnect.\n");
405         return -1;
406     }
407
408     struct timeval endTime;
409     if (gettimeofday(&endTime, NULL) == -1) {
410         perror("appWrite - startup gettimeofday");
411     }
412
413     struct ExecTimes times = { &startTime, &startDataTime, &endDataTime,
414                               &endTime };
415
416     printf("\n\n");
417     printTransmitterReport(totalBytesWritten, fileSize, &times);
418     return 0;
419 }
```

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
13 #define BAUDRATE B38400
14
15 //TODO - Ns and Nr.
16 #define FLAG 0x7E
17 #define F1 0x7D
18 #define F2 0x5E
19 #define A_3 0x03
20 #define A_1 0x01
21 #define C_I 0x0
22 #define C_SET 0x03
23 #define C_UA 0x07
24 #define C_RR 0x05
25 #define C_REJ 0x01
26 #define C_DISC 0xB
27 #define ESC 0x7D
28
29 // S and U frames
30 // A | C | BCC
31 #define A_IND_RESP 0
32 #define C_IND_RESP 1
33 #define BCC_IND_RESP 2
34
35 #define I_FRAMES_SEQ_NUM_BIT(x) (x >> 6)
36 #define S_U_FRAMES_SEQ_NUM_BIT(x) (x >> 7)
37
38 #define TIMEOUT 3 //seconds
```



```
39 #define MAX_TIME_OUTS 5 // attempts
40 #define MAX_REJS 5 //attempts
41
42 static bool timedOut = false;
43 static struct termios oldtio;
44 static enum CommsType global_type;
45
46 void sigAlarmHandler(int sig) {
47     timedOut = true;
48 }
49
50 enum FrameTypeRes {
51     DATA, SET, DISC, UA, RR, REJ, IGNORE, ERROR
52 };
53
54 enum ReadFrameState {
55     AWAITING_FLAG, AWAITING_A, AWAITING_C,
56     // C begin
57     FOUND_I,
58     FOUND_SET,
59     FOUND_DISC,
60     FOUND_UA,
61     FOUND_RR,
62     FOUND_REJ,
63     UNKNOWN_C,
64     // C end
65     VALIDATED_BCC_I,
66     VALIDATED_BCC_OTHERS,
67     READING_I_DATA
68 };
69
70 enum ReadFrameState interpretC(char c) {
71     switch (c & 0x3F) { // ignore sequence number
72     case 0x0:
73         return FOUND_I;
74     case 0x3:
75         return FOUND_SET;
76     case 0xB:
77         return FOUND_DISC;
78     case 0x7:
79         return FOUND_UA;
80     case 0x5:
81         return FOUND_RR;
82     case 0x1:
83         return FOUND_REJ;
84     default:
85         return UNKNOWN_C;
86     }
87 }
88
89 bool validBCC1(char A_BYTE, char C, char BCC1) {
90     return BCC1 == (A_BYTE ^ C);
91 }
92
93 enum FrameTypeRes readFrame(int fd, char **frame, int *frameLength) {
94     *frame = malloc(5);
95     *frameLength = 5;
96     (*frame)[0] = FLAG;
```



```
97  enum ReadFrameState state = AWAITING_FLAG;
98  enum FrameTypeRes frameTypeRes;
99  char buf;
100 int bytesRead;
101 while ((bytesRead = read(fd, &buf, 1)) != -1) {
102     if (bytesRead == 0) {
103         continue;
104     }
105     switch (state) {
106     case AWAITING_FLAG:
107         if (buf == FLAG) {
108             state = AWAITING_A;
109         }
110         break;
111     case AWAITING_A:
112         if (buf != FLAG) {
113             state = AWAITING_C;
114             (*frame)[1] = buf;
115         }
116         break;
117     case AWAITING_C:
118         state = interpretC(buf);
119         (*frame)[2] = buf;
120         break;
121     case UNKNOWN_C:
122         return IGNORE;
123         break;
124     case FOUND_I:
125         state = VALIDATED_BCC_I;
126         (*frame)[3] = buf;
127         if (!validBCC1((*frame)[1], (*frame)[2], (*frame)[3])) {
128             return IGNORE;
129         }
130         break;
131     case FOUND_SET:
132         state = VALIDATED_BCC_OTHERS;
133         frameTypeRes = SET;
134         (*frame)[3] = buf;
135         if (!validBCC1((*frame)[1], (*frame)[2], (*frame)[3])) {
136             return IGNORE;
137         }
138         break;
139     case FOUND_DISC:
140         state = VALIDATED_BCC_OTHERS;
141         frameTypeRes = DISC;
142         (*frame)[3] = buf;
143         if (!validBCC1((*frame)[1], (*frame)[2], (*frame)[3])) {
144             return IGNORE;
145         }
146         break;
147     case FOUND_UA:
148         state = VALIDATED_BCC_OTHERS;
149         frameTypeRes = UA;
150         (*frame)[3] = buf;
151         if (!validBCC1((*frame)[1], (*frame)[2], (*frame)[3])) {
152             return IGNORE;
153         }
154         break;
```



```
155     case FOUND_RR:
156         state = VALIDATED_BCC_OTHERS;
157         frameTypeRes = RR;
158         (*frame)[3] = buf;
159         if (!validBCC1((*frame)[1], (*frame)[2], (*frame)[3])) {
160             return IGNORE;
161         }
162         break;
163     case FOUND_REJ:
164         state = VALIDATED_BCC_OTHERS;
165         frameTypeRes = REJ;
166         (*frame)[3] = buf;
167         if (!validBCC1((*frame)[1], (*frame)[2], (*frame)[3])) {
168             return IGNORE;
169         }
170         break;
171     case VALIDATED_BCC_I:
172         (*frame)[4] = buf;
173         if (buf == FLAG) {
174             return DATA;
175         } else {
176             state = READING_I_DATA;
177         }
178         break;
179     case VALIDATED_BCC_OTHERS:
180         if (buf != FLAG) {
181             return IGNORE;
182         } else {
183             (*frame)[4] = FLAG;
184             return frameTypeRes;
185         }
186     case READING_I_DATA:
187         (*frameLength)++;
188         *frame = realloc(*frame, *frameLength);
189         (*frame)[*frameLength - 1] = buf;
190         if (buf == FLAG) {
191             return DATA;
192         }
193     }
194 }
195 return ERROR;
196 }
197
198 int setupConnection(char port[]) {
199
200     /**
201     Open serial port device for reading and writing and not as controlling tty
202     because we don't want to get killed if linenoise sends CTRL-C.
203     */
204     int fd = open(port, O_RDWR | O_NOCTTY, S_IRWXU | S_IRWXG | S_IRWXO);
205     if (fd < 0) {
206         perror(port);
207         exit(-1);
208     }
209
210     struct termios newtio;
211     if (tcgetattr(fd, &oldtio) == -1) { /* save current port settings */
212         perror("tcgetattr");
```



```
213     exit(-1);
214 }
215
216 bzero(&newtio, sizeof(newtio));
217 newtio.c_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;
218 newtio.c_iflag = IGNPAR;
219 newtio.c_oflag = 0;
220
221 /* set input mode (non-canonical, no echo,...) */
222 newtio.c_lflag = 0;
223
224 newtio.c_cc[VTIME] = 1; /* inter-character timer unused (em 100 ms)*/
225 newtio.c_cc[VMIN] = 0; /* blocking read until 0 chars received */
226
227 tcflush(fd, TCIOFLUSH);
228
229 if (tcsetattr(fd, TCSANOW, &newtio) == -1) {
230     perror("tcsetattr");
231     exit(-1);
232 }
233
234 printf("New termios structure set\n");
235
236 return fd;
237 }
238
239 enum CommsType {
240     TRANSMITTER, RECEIVER
241 };
242
243 /**
244  * setMsg must already be allocated with 5 chars.
245  */
246 void makeSetMsg(char *setMsg) {
247     setMsg[0] = FLAG;
248     setMsg[1] = A_3;
249     setMsg[2] = C_SET;
250     setMsg[3] = A_3 ^ C_SET;
251     setMsg[4] = FLAG;
252 }
253
254 /**
255  * uaMsg must already be allocated with 5 chars.
256  */
257 void makeUaMsg(char *uaMsg) {
258     uaMsg[0] = FLAG;
259     uaMsg[1] = A_3;
260     uaMsg[2] = C_UA;
261     uaMsg[3] = A_3 ^ C_UA;
262     uaMsg[4] = FLAG;
263 }
264
265 int llopenTransmitter(int fd) {
266     int numTimeOuts = 0;
267     char setMsg[5];
268     int setMsgSize = 5;
269     makeSetMsg(setMsg);
270     do {
```



```
271     timedOut = false;
272     if (write(fd, setMsg, setMsgSize) == -1) {
273         perror("llopenTransmitter - write");
274         return -1;
275     }
276     alarm(3);
277     signal(SIGALRM, sigAlarmHandler);
278
279     enum FrameTypeRes res;
280     char *frame = NULL;
281     int frameLength = 0;
282     do {
283         res = readFrame(fd, &frame, &frameLength);
284         free(frame);
285     } while (res != UA && !timedOut);
286
287     if (timedOut) {
288         numTimeOuts++;
289         if (numTimeOuts < MAX_TIME_OUTS) {
290             printf( "%d/%d: Timed out on connection establishment. Retrying.\n",
291                 numTimeOuts, MAX_TIME_OUTS);
292         } else {
293             printf( "%d/%d: Timed out on connection establishment. Exiting.\n",
294                 numTimeOuts, MAX_TIME_OUTS);
295             return -1;
296         }
297     }
298     } while (timedOut && numTimeOuts < MAX_TIME_OUTS);
299
300     return 0;
301 }
302
303 int llopenReceiver(int fd) {
304     char *frame = NULL;
305     int frameLength = 0;
306     while (SET != readFrame(fd, &frame, &frameLength)) {
307     }
308     free(frame);
309
310     char uaMsg[5];
311     int uaMsgSize = 5;
312     makeUaMsg(uaMsg);
313     if (write(fd, uaMsg, uaMsgSize) == -1) {
314         perror("llopenReceiver - write");
315         return -1;
316     }
317
318     return 0;
319 }
320
321 int llopen(char port[], enum CommsType type) {
322     global_type = type;
323     int fd = setupConnection(port);
324     switch (type) {
325     case TRANSMITTER:
326         if (llopenTransmitter(fd) == -1) {
327 #ifdef DEBUG
328             printf("llopen(): llopenTransmitter failed.\n");

```



```
329     #endif
330     return -1;
331 }
332 break;
333 case RECEIVER:
334     if (!llopenReceiver(fd) == -1) {
335 #ifdef DEBUG
336         printf("llopen(): llopenReceiver failed.\n");
337 #endif
338         return -1;
339     }
340     break;
341 }
342 #ifdef DEBUG
343     printf("llopen success.\n");
344 #endif
345     return fd;
346 }
347
348 int makeFrame(char *data, int dataLength, char seqNum, char **frame,
349 int *frameLength) {
350     *frameLength = 4 + dataLength + 2;
351     *frame = malloc(*frameLength);
352     if (*frame == NULL) {
353         perror("makeFrame - malloc");
354         return -1;
355     }
356
357     (*frame)[0] = FLAG;
358     (*frame)[1] = A_3;
359     (*frame)[2] = C_1 | (seqNum << 6);
360     (*frame)[3] = (*frame)[1] ^ (*frame)[2];
361
362     char BCC2 = 0;
363     for (int dataInd = 0, frameInd = 4; dataInd < dataLength; dataInd++, frameInd++) {
364         (*frame)[frameInd] = data[dataInd];
365         BCC2 ^= data[dataInd];
366     }
367
368     (*frame)[*frameLength - 2] = BCC2;
369     (*frame)[*frameLength - 1] = FLAG;
370
371     return 0;
372 }
373
374 /**
375  * Outputs stuffedFrame, allocating it. Its length is greater or equal to the original
376  * frame's length.
377  * Start and stop flags aren't stuffed.
378  */
379 int stuffFrame(char *frame, int frameLength, char **stuffedFrame,
380 int *stuffedFrameLength) {
381     *stuffedFrameLength = frameLength;
382     *stuffedFrame = malloc(*stuffedFrameLength);
383     if (*stuffedFrame == NULL) {
384         printf("stuffFrame - malloc");
385         return -1;
386     }
```




```
387
388     (*stuffedFrame)[0] = FLAG;
389     for (int unstuffedInd = 1, stuffedInd = 1; unstuffedInd < frameLength - 1;
390          unstuffedInd++, stuffedInd++) {
391         switch (frame[unstuffedInd]) {
392             case FLAG:
393                 (*stuffedFrameLength)++;
394                 *stuffedFrame = realloc(*stuffedFrame, *stuffedFrameLength);
395                 if (*stuffedFrame == NULL) {
396                     perror("stuffFrame - realloc");
397                     return -1;
398                 }
399                 (*stuffedFrame)[stuffedInd++] = ESC;
400                 (*stuffedFrame)[stuffedInd] = FLAG ^ 0x20;
401                 break;
402             case ESC:
403                 (*stuffedFrameLength)++;
404                 *stuffedFrame = realloc(*stuffedFrame, *stuffedFrameLength);
405                 if (*stuffedFrame == NULL) {
406                     perror("stuffFrame - realloc");
407                     return -1;
408                 }
409                 (*stuffedFrame)[stuffedInd++] = ESC;
410                 (*stuffedFrame)[stuffedInd] = ESC ^ 0x20;
411                 break;
412             default:
413                 (*stuffedFrame)[stuffedInd] = frame[unstuffedInd];
414         }
415     }
416     (*stuffedFrame)[*stuffedFrameLength - 1] = FLAG;
417     return 0;
418 }
419
420 int unstuffFrame(char *stuffedFrame, int stuffedFrameLength, char **frame,
421 int *frameLength) {
422     *frameLength = stuffedFrameLength;
423     *frame = malloc(*frameLength);
424     if (*frame == NULL) {
425         perror("unstuffFrame - malloc");
426         return -1;
427     }
428
429     (*frame)[0] = FLAG;
430     for (int stuffedInd = 1, unstuffedInd = 1; stuffedInd < stuffedFrameLength - 1;
431          stuffedInd++, unstuffedInd++) {
432         switch (stuffedFrame[stuffedInd]) {
433             case ESC:
434                 (*frameLength)--;
435                 *frame = realloc(*frame, *frameLength);
436                 stuffedInd++;
437                 (*frame)[unstuffedInd] = stuffedFrame[stuffedInd] ^ 0x20;
438                 break;
439             default:
440                 (*frame)[unstuffedInd] = stuffedFrame[stuffedInd];
441                 break;
442         }
443     }
444     (*frame)[*frameLength - 1] = FLAG;
```



```
445     return 0;
446 }
447
448 int extractPacket(char **packet, int *packetLength, char *frame, int frameLength) {
449     *packetLength = -4 + frameLength - 2;
450     *packet = malloc(*packetLength);
451     if (*packet == NULL) {
452         perror("extractPacket - malloc");
453         return -1;
454     }
455     memcpy(*packet, frame + 4, *packetLength);
456     return 0;
457 }
458
459 bool validPacketBCC(char *packet, int packetLength, char BCC2) {
460     char acc = 0;
461     for (int i = 0; i < packetLength; i++) {
462         acc ^= packet[i];
463     }
464     return BCC2 == acc;
465 }
466
467 bool frameIsDuplicated(char *frame, char previousSeqNum) {
468     if (previousSeqNum == -1) {
469         return false;
470     }
471     return previousSeqNum == (I_FRAMES_SEQ_NUM_BIT(frame[2]));
472 }
473
474 int sendReady(int fd, char seqNumber) {
475     int responseSize = 5;
476     char response[responseSize];
477
478     bzero(response, responseSize);
479
480     response[0] = FLAG;
481     response[1] = A_3;
482     response[2] = C_RR | (seqNumber << 7);
483     response[3] = A_3 ^ (C_RR | (seqNumber << 7));
484     response[4] = FLAG;
485
486     if (write(fd, response, responseSize) == -1) {
487         printf("sendReady(): write failed\n");
488         return -1;
489     }
490
491     return 0;
492 }
493
494 int sendRejection(int fd, char seqNumber) {
495     int responseSize = 5;
496     char response[responseSize];
497
498     bzero(response, responseSize);
499
500     response[0] = FLAG;
501     response[1] = A_3;
502     response[2] = C_REJ | (seqNumber << 7);
```



```
503     response[3] = A_3 ^ (C_REJ | (seqNumber << 7));
504     response[4] = FLAG;
505
506     if (write(fd, response, responseSize) == -1) {
507         printf("sendReady(): write failed\n");
508         return -1;
509     }
510     return 0;
511 }
512
513 /**
514  * @return Buffer length (bytes read), -1 if error.
515  */
516 int llread(int fd, char **packet) {
517     static char previousSeqNum = -1;
518     bool rejected = false;
519     int numRejects = 0;
520     *packet = NULL;
521     int packetLength = 0;
522
523     char frameC = 0;
524     bool discardedPacket = false;
525
526     do {
527         if (*packet != NULL) {
528             free(*packet);
529             *packet = NULL;
530         }
531         discardedPacket = false;
532         rejected = false;
533         char *stuffedFrame = NULL;
534         int stuffedFrameLength = 0;
535         while (true) {
536             enum FrameTypeRes res
537                 = readFrame(fd, &stuffedFrame, &stuffedFrameLength);
538             if (res == DATA) {
539                 break;
540             } else if (res == IGNORE) {
541                 return 0;
542             }
543             free(stuffedFrame);
544         }
545
546         char *frame = NULL;
547         int frameLength = 0;
548         if (unstuffFrame(stuffedFrame, stuffedFrameLength, &frame, &frameLength) == -1) {
549             #ifdef DEBUG
550                 printf("llread(): unstuffFrame failed.\n");
551             #endif
552             free(stuffedFrame);
553             return -1;
554         }
555         free(stuffedFrame);
556         frameC = frame[2];
557
558         if (extractPacket(packet, &packetLength, frame, frameLength) == -1) {
559             #ifdef DEBUG
560                 printf("llread(): extractPacket failed.\n");
```



```
561     #endif
562     return -1;
563 }
564
565     char BCC2 = frame[frameLength - 2];
566     if (!validPacketBCC(*packet, packetLength, BCC2)) {
567         if (framesDuplicated(frame, previousSeqNum)) {
568             sendReady(fd, !_FRAMES_SEQ_NUM_BIT(frameC));
569             rejected = false;
570         } else {
571             sendRejection(fd, !_FRAMES_SEQ_NUM_BIT(frameC));
572             rejected = true;
573         }
574     } else {
575         if (framesDuplicated(frame, previousSeqNum)) {
576             free(*packet);
577             *packet = NULL;
578             discardedPacket = true;
579         }
580         sendReady(fd, !_FRAMES_SEQ_NUM_BIT(frameC));
581         rejected = false;
582     }
583
584     free(frame);
585
586     if (rejected) {
587         numRejects++;
588 #ifdef DEBUG
589         printf("llread(): Packet rejected.\n");
590 #endif
591     }
592     } while (rejected && numRejects < MAX_REJS);
593     previousSeqNum = !_FRAMES_SEQ_NUM_BIT(frameC);
594
595     if (!rejected && !discardedPacket) {
596         return packetLength;
597     } else {
598         return 0;
599     }
600 }
601
602 /**
603  * @return Bytes written, -1 if error.
604  */
605 int llwrite(int fd, char *data, int dataLength) {
606     static char seqNum = 0;
607     char *frame = NULL;
608     int frameLength = 0;
609     char *stuffedFrame = NULL;
610     int stuffedFrameLength = 0;
611     char *responseFrame = NULL;
612     int responseFrameLength = 0;
613     int numTimeOuts = 0;
614     int numRejects = 0;
615     bool accepted = false;
616
617     makeFrame(data, dataLength, seqNum, &frame, &frameLength);
618     stuffFrame(frame, frameLength, &stuffedFrame, &stuffedFrameLength);
```



```
619     free(frame);
620     do {
621         timedOut = false;
622         accepted = true;
623         if (write(fd, stuffedFrame, stuffedFrameLength) == -1) {
624             perror("llwrite - write");
625             free(stuffedFrame);
626             return -1;
627         }
628         alarm(3);
629         signal(SIGALRM, sigAlarmHandler);
630         enum FrameTypeRes res;
631         bool endRead = true;
632         do {
633             res = readFrame(fd, &responseFrame, &responseFrameLength);
634             endRead = true;
635             switch (res) {
636                 case RR:
637                     accepted = true;
638                     seqNum = S_U_FRAMES_SEQ_NUM_BIT(responseFrame[2]);
639                     break;
640                 case REJ:
641                     accepted = false;
642                     numRejects++;
643                     break;
644                 case IGNORE:
645                     accepted = true;
646                     break;
647                 case ERROR:
648                     timedOut = true;
649                     break;
650                 default:
651                     endRead = false;
652             }
653             free(responseFrame);
654         } while (!timedOut && !endRead);
655
656         alarm(0);
657
658         if (timedOut) {
659             ++numTimeOuts;
660             if (numTimeOuts < MAX_TIME_OUTS) {
661                 printf("%d/%d: Timed out while sending packet. Retrying.\n",
662                     numTimeOuts, MAX_TIME_OUTS);
663             } else {
664                 printf("%d/%d: Timed out while sending packet. Exiting.\n",
665                     numTimeOuts, MAX_TIME_OUTS);
666                 alarm(0);
667                 free(stuffedFrame);
668                 return -1;
669             }
670         }
671     } while ((timedOut && numTimeOuts < MAX_TIME_OUTS)
672         || (!accepted && numRejects < MAX_REJS));
673
674     free(stuffedFrame);
675
676     if (numTimeOuts >= MAX_TIME_OUTS) {
```



```
677     return 0;
678 } else {
679     return dataLength;
680 }
681 }
682
683 /**
684  * discMsg must already be allocated with 5 chars.
685  */
686 void makeTransDiscMsg(char *discMsg) {
687     discMsg[0] = FLAG;
688     discMsg[1] = A_3;
689     discMsg[2] = C_DISC;
690     discMsg[3] = A_3 ^ C_DISC;
691     discMsg[4] = FLAG;
692 }
693
694 /**
695  * discMsg must already be allocated with 5 chars.
696  */
697 void makeReceiverDiscMsg(char *discMsg) {
698     discMsg[0] = FLAG;
699     discMsg[1] = A_1;
700     discMsg[2] = C_DISC;
701     discMsg[3] = A_1 ^ C_DISC;
702     discMsg[4] = FLAG;
703 }
704
705 void makeTransUaMsg(char *uaMsg) {
706     uaMsg[0] = FLAG;
707     uaMsg[1] = A_1;
708     uaMsg[2] = C_UA;
709     uaMsg[3] = A_1 ^ C_UA;
710     uaMsg[4] = FLAG;
711 }
712
713 /**
714  * Returns 1 if success, -1 if error.
715  */
716 int llcloseTransmitter(int fd) {
717     int numTimeOuts = 0;
718     int discMsgSize = 5;
719     char discMsg[5];
720     makeTransDiscMsg(discMsg);
721     do {
722         timedOut = false;
723         if (write(fd, discMsg, discMsgSize) == -1) {
724             perror("llcloseTransmitter - write");
725             return -1;
726         }
727         alarm(3);
728         signal(SIGALRM, sigAlarmHandler);
729
730         enum FrameTypeRes res;
731         char *frame = NULL;
732         int frameLength = 0;
733         do {
734             res = readFrame(fd, &frame, &frameLength);
```



```
735     free(frame);
736 } while (res != DISC && !timedOut);
737
738 alarm(0);
739
740 if (timedOut) {
741     numTimeOuts++;
742     if (numTimeOuts < MAX_TIME_OUTS) {
743         printf("%d/%d: Timed out on disconnection. Retrying.\n",
744             numTimeOuts, MAX_TIME_OUTS);
745     } else {
746         printf("%d/%d: Timed out on disconnection. Exiting.\n",
747             numTimeOuts, MAX_TIME_OUTS);
748         return -1;
749     }
750 }
751 } while (timedOut && numTimeOuts < MAX_TIME_OUTS);
752
753 int uaMsgSize = 5;
754 char uaMsg[5];
755 makeTransUaMsg(uaMsg);
756 if (write(fd, uaMsg, uaMsgSize) == -1) {
757     perror("llcloseTransmitter - write");
758     return -1;
759 }
760
761 return 1;
762 }
763
764 /**
765  * Returns 1 if success, -1 if error.
766  */
767 int llcloseReceiver(int fd) {
768     enum FrameTypeRes res;
769     char *frame = NULL;
770     int frameLength = 0;
771     do {
772         res = readFrame(fd, &frame, &frameLength);
773         free(frame);
774     } while (res != DISC);
775
776     int discMsgSize = 5;
777     char discMsg[5];
778     makeReceiverDiscMsg(discMsg);
779
780     int numTimeOuts = 0;
781     do {
782         timedOut = false;
783         if (write(fd, discMsg, discMsgSize) == -1) {
784             perror("llcloseReceiver - write");
785             return -1;
786         }
787         alarm(3);
788         signal(SIGALRM, sigAlarmHandler);
789
790         enum FrameTypeRes res;
791         char *frame = NULL;
792         int frameLength = 0;
```



```
793     do {
794         res = readFrame(fd, &frame, &frameLength);
795         free(frame);
796     } while (res != UA && !timedOut);
797
798     alarm(0);
799
800     if (timedOut) {
801         numTimeOuts++;
802         if (numTimeOuts < MAX_TIME_OUTS) {
803             printf( "%d/%d: Timed out on disconnection acknowledgement. Retrying.\n",
804                 numTimeOuts, MAX_TIME_OUTS);
805         } else {
806             printf( "%d/%d: Timed out on disconnection acknowledgement. Exiting.\n",
807                 numTimeOuts, MAX_TIME_OUTS);
808             return -1;
809         }
810     }
811     } while (timedOut && numTimeOuts < MAX_TIME_OUTS);
812
813     return 1;
814 }
815
816 /**
817  * Returns 1 if success, -1 if error.
818  */
819 int llclose(int fd) {
820     switch (global_type) {
821         case TRANSMITTER:
822             return llcloseTransmitter(fd);
823         case RECEIVER:
824             return llcloseReceiver(fd);
825         default:
826             #ifdef DEBUG
827                 printf("llclose(): Invalid CommsType.\n");
828             #endif
829             return -1;
830     }
831 }
```

noncanonical.c

```
1  /*Non-Canonical Input Processing*/
2
3  #include "appAPI.h"
4
5  #define _POSIX_SOURCE 1 /* POSIX compliant source */
6  #define FALSE 0
7  #define TRUE 1
8
9  volatile int STOP = FALSE;
10
11 int main(int argc, char** argv) {
12     #ifdef DEBUG
13         printf("Debug mode: ON.\n");
14     #endif
15
16     if (argc < 2) {
```




```
17     printf("Usage:\tSerialPort\n\ttex: /dev/ttyS0\n");
18     exit(1);
19 }
20
21 if (appRead(argv[1]) == -1) {
22     printf("appRead() failed.\n");
23     return -1;
24 }
25
26 return 0;
27 }
```

writenoncanonical.c

```
1  /*Non-Canonical Input Processing*/
2
3  #include "appAPI.h"
4
5  #define MODEMDEVICE "/dev/ttyS1"
6  #define _POSIX_SOURCE 1 /* POSIX compliant source */
7  #define FALSE 0
8  #define TRUE 1
9
10 volatile int STOP = FALSE;
11
12 int main(int argc, char** argv) {
13     if (argc < 2) {
14         printf("Usage:\tSerialPort FileName\n");
15         printf("\tex: /dev/ttyS0 image.png\n");
16         exit(1);
17     }
18
19     if (appWrite(argv[1], argv[2]) == -1) {
20         printf("appWrite() failed.\n");
21         return -1;
22     }
23
24     return 0;
25 }
```