

CSC258 Final Report: The Morse Code Decoder

Gramano, Francesco Shah, Milind

April 5, 2013

Introduction

We chose to implement a Morse Code Decoder for a few reasons. The first of these reasons was to trace what might now be considered as a primitive beginning to radio communications. We wanted to replicate a self-sufficient version of these means of communication on the DE2 board to revisit the humble beginning of the use of Morse Code as an encoding of the alphabet. More importantly, we wanted to study the ramifications of requiring the synchronization of a finite state machine with the clock of the DE2 board in this process of replicating a Morse Code interpreter.

Methods

The first problem was to record the duration of a press of *KEY*[1] on the DE2 board, since the differentiation of a dot ('.') and a dash ('-') in Morse Code is relative to the amount of time that the key is held. Of course, since older standards of Morse Code are much faster than can be provided as input with the press of *KEY*[1], we used the convention of airway beacons where the shortest unit of time (the dot duration) is 0.5 seconds. The remaining possible inputs from pressing or the absence of pressing the key are built relatively from the dot duration. So, a dash is interpreted after three dot durations (or 1.5 seconds), a character break is interpreted after three dot durations of not pressing the key, and a word break (a space between successive characters) is interpreted after seven dot durations (or 3.5 seconds)

of not pressing the key. Then, a character becomes a concatenation of dots and or dashes until either a character break or word break is reached. The duration of the press of $KEY[1]$ was monitored synchronously with the clock by counting the number of cycles that the key had either been pressed or had not been pressed for and computing its duration by dividing the total number of cycles for either by the frequency of the clock of the DE2 board (50 MHz). Since an operator of the Morse Code interpreter could not guarantee that they would, say, hold $KEY[1]$ for exactly 75 million pulses of the clock in order to register a dot, durations were approximated over a range of time from the exact number, with an allowed error of 0.2 seconds for each interval. This is reflected in the code in the appendix on lines 114, 115, 133, 134.

Refer to the following *Huffman Coding* diagram of Morse Code for the representations of letters as sequences of dots and or dashes. Note that a traversal towards the left is equivalent to concatenating a dot to a sequence, a traversal towards the right is equivalent to concatenating a dash to a sequence, and the **start** state is represented by the empty string which contains no dots or dashes.

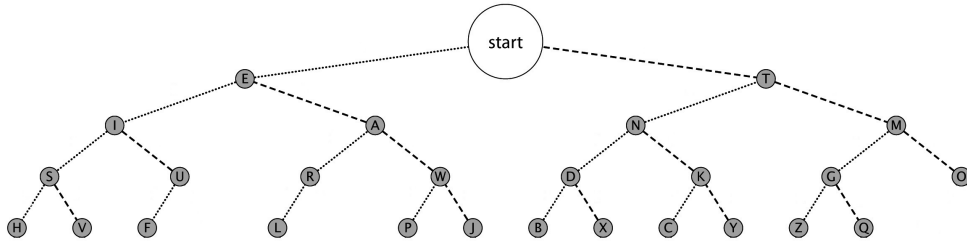


Figure 1: A *Huffman Coding* diagram of the representation of Morse Code. This figure is a simplification of the *Huffman Coding* diagram for international Morse Code found at http://en.wikipedia.org/wiki/Morse_code

In order to restrict the input to the sequences which represent the letters of the alphabet, the characters which have the longest sequence and can therefore not transition to another character (since having the longest sequence would mean that concatenating another dot or dash would result in a non-existent letter) were made to transition to a **BREAK** state which was also overloaded to represent a character break and a word break.

[**Note:** since a copy of the finite state machine used for transitioning between characters cannot fit between the margins of an 8.5 by 11 inches sheet of paper, the reader will have to interpret the FSM implicitly from **Figure 1** and the description of the **BREAK** state. The FSM has 27 states: one for each letter and a **BREAK** state. A shortened copy of the FSM can still be found in the appendix.]

After interpreting a word break or a character break, the letter of the alphabet resulting from the cumulated sequence gets printed on the LCD display of the DE2 board and the current display gets shifted over by one character. Effectively, the LCD display acts as a shift register.

Results

Overloading the **BREAK** state turned out to be a greater complication than expected because of the different transitions that could lead to it, and the specific results that should accommodate each of them. Also, designing the FSM turned out to be more difficult than expected because of the large number of cases, each of which must evidently be a correct translation of the previously shown *Huffman Coding* diagram.

Discussion

Originally, an attempt was made at simplifying the recording of input by discretizing the moments where the key-press would be monitored. It was initially decided that, with a similar acceptable error of 0.2 seconds per dot duration, the checking of the key-press would only be monitored at every approximate dot duration. However, since this model would equate three dots with one dash, we eventually realized this would result in an overlap of representations. For example, consider how the letter T is represented by one dash. Discretizing the checking of input, as was just described, would have it get represented by three consecutive dots, but, this matches the representation of the letter S, so this method of interpretation does not produce unique sequences for characters. That said, we defaulted to counting the number of cycles that *KEY*[1] would have **continuously** been held or not been held for.

Similarly, we initially attempted to enumerate the states that represent the characters as a traversal of the *Huffman Coding* diagram as a binary tree, where a traversal to a left child would concatenate a 0 and a traversal to a right child would concatenate a 1. However, since we were only able to store these as binary numbers and not strings, leading zeroes became indistinguishable and state representations were not unique. For example, consider how state *E* would be enumerated as 0, and state *I* would be enumerated as 00. Treating these representations as numbers, they get treated the same. We solved this problem by enumerating the states by a level-order traversal of the *Huffman Coding* diagram as a binary tree, starting from 1. With this method, state *E* was 1, state *T* was 2, and so on and so forth. This is visible towards the beginning of the code in the appendix (on lines 37 to 43 of `morsecode.v`).

The Morse Code interpreter was mostly functional; however, although the dot duration was set to be 0.5 seconds, it was still somewhat difficult to hold `KEY[1]` for the proper durations in order to represent the desired character. This also made it much harder to verify. Though the idea of using a range of time to approximate the durations of each kind of input was practical and helpful, the approximate durations were too small, especially for any beginner of Morse Code.

We learned that though some of the Morse Code sequences of characters are prefixes of sequences of other characters and monopolizing on this property to create a finite state machine would be elegant, it turns out to be more of a complication. Synchronizing the finite state machine with the LCD display and the clock of the DE2 board becomes a hassle since the transitions of the characters that have the longest accepted sequences (the characters which are represented as leaves on the previous *Huffman Coding* diagram) need to be treated differently from the other characters. This is reflected in the code in the appendix where the reader will see `'#1000'`, which was used at certain points to delay certain transitions of the FSM from executing the remainder of the `always` block that surrounded it.

To make it easier for an operator to get used to the device, we would let them specify their average input speed with the switches at execution. With their average speed (in words per minute) we would calculate the dot

duration, T (in milliseconds), with the formula $T = \frac{1200}{W}$, where they would be specifying W . Approximate ranges surrounding each duration could still be used and the ranges and remaining durations would be calculated relatively from the new dot duration, as per their relative lengths stated earlier.

Conclusion

Through the process of constructing the Morse Code interpreter we understood that the simplicity of an earlier version was only meant to be decoded by a human operator; though, using a finite state machine to interpret Morse Code was a good exercise of the use of Verilog and the synchronization to the clock. With the previously noted extension of giving the operator the ability to specify their own input speed, this method of automating the interpretation of Morse Code could be found practical, especially for a beginner who would otherwise have to look up the Morse Code sequence of every character.

Appendix

The code of the main module:

morsecode.v

```
1 module morsecode(  
2     input  CLOCK_50,      //50 MHz clock  
3     input  [1:0] KEY,     //KEY1 is the morse key  
4     inout  [35:0] GPIO_0,GPIO_1,  
5     output LCD_ON,  
6     output LCD_BLON,  
7     output LCD_RW,  
8     output LCD_EN,  
9     output LCD_RS,  
10    input  PS2_DAT,  
11    input  PS2_CLK,  
12    inout  [7:0] LCD_DATA,  
13    output [7:0] LEDG  
14 );  
15  
16    reg [31:0] CCount; //Clock cycle count  
17    reg [31:0] Kcount; //Key-press clock cycle count  
18    reg [31:0] Bcount; //Inactivity clock cycle count  
19  
20    reg w; //w == 1 --> dash, w == 0 --> dot  
21    reg v; //v == 0 --> c break, v == 1 --> w break  
22  
23    reg newdot; //whether there is a new dot/dash  
24    reg newbreak; //whether there is a new break  
25  
26    wire reset;  
27    wire ore;  
28    reg pulse;  
29  
30    reg [7:0] letter_one; //for immediate letter of  
    the first row  
31    reg [7:0] letter_two; //for immediate letter of  
    the second row
```

```

32 reg [127:0] letters_LCD_one; //holds the values of
    the first row of the LCD
33 reg [127:0] letters_LCD_two; //holds the values of
    the second row of the LCD
34
35 //State enumeration for character FSM
36 parameter
37 A = 5'd4, B = 5'd21, C = 5'd23, D = 5'd11,
38 E = 5'd1, F = 5'd17, G = 5'd13, H = 5'd15,
39 I = 5'd3, J = 5'd20, K = 5'd12, L = 5'd18,
40 M = 5'd6, N = 5'd5, O = 5'd14, P = 5'd19,
41 Q = 5'd26, R = 5'd9, S = 5'd7, T = 5'd2,
42 U = 5'd8, V = 5'd16, W = 5'd10, X = 5'd22,
43 Y = 5'd24, Z = 5'd25, BREAK = 4'd0;
44
45 //The Characters' values for the LCD
46 parameter
47 LA = 8'h41, LB = 8'h42, LC = 8'h43, LD = 8'h44,
48 LE = 8'h45, LF = 8'h46, LG = 8'h47, LH = 8'h48,
49 LI = 8'h49, LJ = 8'h4A, LK = 8'h4B, LL = 8'h4C,
50 LM = 8'h4D, LN = 8'h4E, LO = 8'h4F, LP = 8'h50,
51 LQ = 8'h51, LR = 8'h52, LS = 8'h53, LT = 8'h54,
52 LU = 8'h55, LV = 8'h56, LW = 8'h57, LX = 8'h58,
53 LY = 8'h59, LZ = 8'h5A, DOT = 8'h2E, DASH = 8'h2D,
54 SPACE = 8'h20;
55
56 reg [4:0] y_Q, Y_D; /* y_Q == current state,
57 Y_D == next state */
58
59 //Set defaults
60 initial
61 begin
62     y_Q = BREAK;
63     Y_D = BREAK;
64     letters_LCD_two[7:0] = SPACE;
65     letters_LCD_two[15:8] = SPACE;
66     letters_LCD_two[23:16] = SPACE;
67     letters_LCD_two[31:24] = SPACE;

```

```

68     letters_LCD_two[39:32] = SPACE;
69     letters_LCD_two[47:40] = SPACE;
70     letters_LCD_two[55:48] = SPACE;
71     letters_LCD_two[63:56] = SPACE;
72     letters_LCD_two[71:64] = SPACE;
73     letters_LCD_two[79:72] = SPACE;
74     letters_LCD_two[87:80] = SPACE;
75     letters_LCD_two[95:88] = SPACE;
76     letters_LCD_two[103:96] = SPACE;
77     letters_LCD_two[111:104] = SPACE;
78     letters_LCD_two[119:112] = SPACE;
79     letters_LCD_two[127:120] = SPACE;
80
81     letters_LCD_one[7:0] = SPACE;
82     letters_LCD_one[15:8] = SPACE;
83     letters_LCD_one[23:16] = SPACE;
84     letters_LCD_one[31:24] = SPACE;
85     letters_LCD_one[39:32] = SPACE;
86     letters_LCD_one[47:40] = SPACE;
87     letters_LCD_one[55:48] = SPACE;
88     letters_LCD_one[63:56] = SPACE;
89     letters_LCD_one[71:64] = SPACE;
90     letters_LCD_one[79:72] = SPACE;
91     letters_LCD_one[87:80] = SPACE;
92     letters_LCD_one[95:88] = SPACE;
93     letters_LCD_one[103:96] = SPACE;
94     letters_LCD_one[111:104] = SPACE;
95     letters_LCD_one[119:112] = SPACE;
96     letters_LCD_one[127:120] = SPACE;
97 end
98
99 always @ (posedge CLOCK_50) begin
100     if (CCount >= 25000000) begin
101         CCount = 0;
102         pulse = ~pulse;
103     end else begin
104         CCount = CCount + 1;
105     end

```



```

106
107     newbreak = 0;
108     newdot = 0;
109
110     if (!KEY[1]) begin
111         Kcount = Kcount + 1;
112         if (Bcount > 0) begin
113             if (Bcount > 15000000) begin
114                 if (Bcount > 91000000) begin
115                     if (Bcount >= 4294967196) begin
116                         Bcount = 0; //reset
117                     end else begin
118                         v = 1; //word break
119                         newbreak = 1;
120                     end
121                 end else begin
122                     v = 0; //character break
123                     newbreak = 1;
124                 end
125             end
126         end
127         Bcount = 0;
128     end
129     if (KEY[1]) begin
130         Bcount = Bcount + 1;
131         if (Kcount > 0) begin
132             if (Kcount > 15000000) begin
133                 if (Kcount > 39000000) begin
134                     if (Kcount >= 4294967196) begin
135                         Kcount = 0; //reset
136                     end else begin
137                         w = 1; //dash
138                         newdot = 1;
139                     end
140                 end else begin
141                     w = 0; //dot
142                     newdot = 1;
143                 end

```

```

144         end
145     end
146     Kcount = 0;
147 end
148 end
149
150 assign LEDG[7:0] = {~pulse,pulse,~pulse,pulse,~
151     pulse,pulse,~pulse,pulse};
152 assign ore = newdot ^ newbreak;
153
154 /* Due to the nature of the previous loop and how
155 it affects newdot and newbreak, they will never
156 be the same so we can set
157 ore = newdot || newbreak, but it is left as a
158 xor just to be clear that we're avoiding
159 unsteady behaviour nonetheless */
160
161 always @(posedge ore)
162     begin: state_table
163     case (newdot)
164     1: begin
165     case (y_Q)
166     BREAK:
167         if (w) begin
168             Y_D = T;
169             letter_two = DASH;
170             y_Q = Y_D;
171
172             //Shift a dash into the second row
173             letters_LCD_two = letters_LCD_two >> 8;
174             letters_LCD_two[127:120] = letter_two;
175             #1000; //prevent completion of the block
176         end
177     else begin
178         Y_D = E;
179         letter_two = DOT;
180         y_Q = Y_D;

```

```

181
182      //Shift a dot into the second row
183      letters_LCD_two = letters_LCD_two >> 8;
184      letters_LCD_two[127:120] = letter_two;
185      #1000; //prevent completion of the block
186  end
187
188  E:
189      if (w) begin
190          Y_D = A;
191          letter_two = DASH;
192          y_Q = Y_D;
193
194          //Shift a dash into the second row
195          letters_LCD_two = letters_LCD_two >> 8;
196          letters_LCD_two[127:120] = letter_two;
197          #1000; //prevent completion of the block
198      end
199      else begin
200          Y_D = I;
201          letter_two = DOT;
202          y_Q = Y_D;
203
204          //Shift a dot into the second row
205          letters_LCD_two = letters_LCD_two >> 8;
206          letters_LCD_two[127:120] = letter_two;
207          #1000; //prevent completion of the block
208      end
209
210  T:
211      if (w) begin
212          Y_D = M;
213          letter_two = DASH;
214          y_Q = Y_D;
215
216          //Shift a dash into the second row
217          letters_LCD_two = letters_LCD_two >> 8;
218          letters_LCD_two[127:120] = letter_two;

```

```

219     end
220     else begin
221         Y_D = N;
222         letter_two = DOT;
223         y_Q = Y_D;
224
225         //Shift a dot into the second row
226         letters_LCD_two = letters_LCD_two >> 8;
227         letters_LCD_two[127:120] = letter_two;
228         #1000; //prevent completion of the block
229     end
230
231 I:
232     if (w) begin
233         Y_D = U;
234         letter_two = DASH;
235         y_Q = Y_D;
236
237         //Shift a dash into the second row
238         letters_LCD_two = letters_LCD_two >> 8;
239         letters_LCD_two[127:120] = letter_two;
240         #1000; //prevent completion of the block
241     end
242     else begin
243         Y_D = S;
244         letter_two = DOT;
245         y_Q = Y_D;
246
247         //Shift a dot into the second row
248         letters_LCD_two = letters_LCD_two >> 8;
249         letters_LCD_two[127:120] = letter_two;
250         #1000; //prevent completion of the block
251     end
252
253 A:
254     if (w) begin
255         Y_D = W;
256         letter_two = DASH;

```

```

257         y_Q = Y_D;
258
259         //Shift a dash into the second row
260         letters_LCD_two = letters_LCD_two >> 8;
261         letters_LCD_two[127:120] = letter_two;
262         #1000; //prevent completion of the block
263     end
264     else begin
265         Y_D = R;
266         letter_two = DOT;
267         y_Q = Y_D;
268
269         //Shift a dot into the second row
270         letters_LCD_two = letters_LCD_two >> 8;
271         letters_LCD_two[127:120] = letter_two;
272         #1000; //prevent completion of the block
273     end
274
275 N:
276     if (w) begin
277         Y_D = K;
278         letter_two = DASH;
279         y_Q = Y_D;
280
281         //Shift a dash into the second row
282         letters_LCD_two = letters_LCD_two >> 8;
283         letters_LCD_two[127:120] = letter_two;
284         #1000; //prevent completion of the block
285     end
286     else begin
287         Y_D = D;
288         letter_two = DOT;
289         y_Q = Y_D;
290
291         //Shift a dot into the second row
292         letters_LCD_two = letters_LCD_two >> 8;
293         letters_LCD_two[127:120] = letter_two;
294         #1000; //prevent completion of the block

```

```

295         end
296
297     M:
298         if (w) begin
299             Y_D = 0;
300             letter_two = DASH;
301             y_Q = Y_D;
302
303             //Shift a dash into the second row
304             letters_LCD_two = letters_LCD_two >> 8;
305             letters_LCD_two[127:120] = letter_two;
306             #1000; //prevent completion of the block
307         end
308         else begin
309             Y_D = G;
310             letter_two = DOT;
311             y_Q = Y_D;
312
313             //Shift a dot into the second row
314             letters_LCD_two = letters_LCD_two >> 8;
315             letters_LCD_two[127:120] = letter_two;
316             #1000; //prevent completion of the block
317         end
318
319     S:
320         if (w) begin
321             Y_D = V;
322             letter_one = LV;
323             letter_two = DASH;
324             y_Q = Y_D;
325
326             //Shift a dash into the second row
327             letters_LCD_two = letters_LCD_two >> 8;
328             letters_LCD_two[127:120] = letter_two;
329             #1000; //prevent completion of the block
330         end
331         else begin
332             Y_D = H;

```

```

333         letter_one = LH;
334         letter_two = DOT;
335         y_Q = Y_D;
336
337         //Shift a dot into the second row
338         letters_LCD_two = letters_LCD_two >> 8;
339         letters_LCD_two[127:120] = letter_two;
340         #1000; //prevent completion of the block
341     end
342
343 U:
344     if (!w) begin
345         Y_D = F;
346         letter_one = LF;
347         letter_two = DOT;
348         y_Q = Y_D;
349
350         //Shift a dot into the second row
351         letters_LCD_two = letters_LCD_two >> 8;
352         letters_LCD_two[127:120] = letter_two;
353     end
354     else begin
355         Y_D = BREAK;
356         letter_one = LU;
357         y_Q = Y_D;
358         #1000; //prevent completion of the block
359     end
360
361 R:
362     if (!w) begin
363         Y_D = L;
364         letter_one = LL;
365         letter_two = DOT;
366         y_Q = Y_D;
367
368         //Shift a dot into the second row
369         letters_LCD_two = letters_LCD_two >> 8;
370         letters_LCD_two[127:120] = letter_two;

```

```

371         #1000; //prevent completion of the block
372     end
373     else begin
374         Y_D = BREAK;
375         letter_one = LR;
376         y_Q = Y_D;
377     end
378
379 W:
380     if (w) begin
381         Y_D = J;
382         letter_one = LJ;
383         letter_two = DASH;
384         y_Q = Y_D;
385
386         //Shift a dash into the second row
387         letters_LCD_two = letters_LCD_two >> 8;
388         letters_LCD_two[127:120] = letter_two;
389         #1000; //prevent completion of the block
390     end
391     else begin
392         Y_D = P;
393         letter_one = LP;
394         letter_two = DOT;
395         y_Q = Y_D;
396
397         //Shift a dot into the second row
398         letters_LCD_two = letters_LCD_two >> 8;
399         letters_LCD_two[127:120] = letter_two;
400         #1000; //prevent completion of the block
401     end
402
403 D:
404     if (w) begin
405         Y_D = X;
406         letter_one = LX;
407         letter_two = DASH;
408         y_Q = Y_D;

```



```

409
410      //Shift a dash into the second row
411      letters_LCD_two = letters_LCD_two >> 8;
412      letters_LCD_two[127:120] = letter_two;
413  end
414  else begin
415      Y_D = B;
416      letter_one = LB;
417      letter_two = DOT;
418      y_Q = Y_D;
419
420      //Shift a dot into the second row
421      letters_LCD_two = letters_LCD_two >> 8;
422      letters_LCD_two[127:120] = letter_two;
423  end
424
425  K:
426      if (w) begin
427          Y_D = Y;
428          letter_one = LY;
429          letter_two = DASH;
430          y_Q = Y_D;
431
432          //Shift a dash into the second row
433          letters_LCD_two = letters_LCD_two >> 8;
434          letters_LCD_two[127:120] = letter_two;
435      end
436      else begin
437          Y_D = C;
438          letter_one = LC;
439          letter_two = DOT;
440          y_Q = Y_D;
441
442          //Shift a dot into the second row
443          letters_LCD_two = letters_LCD_two >> 8;
444          letters_LCD_two[127:120] = letter_two;
445      end
446

```

```

447 G:
448     if (w) begin
449         Y_D = Q;
450         letter_one = LQ;
451         letter_two = DASH;
452         y_Q = Y_D;
453
454         //Shift a dash into the second row
455         letters_LCD_two = letters_LCD_two >> 8;
456         letters_LCD_two[127:120] = letter_two;
457     end
458     else begin
459         Y_D = Z;
460         letter_one = LZ;
461         letter_two = DOT;
462         y_Q = Y_D;
463
464         //Shift a dot into the second row
465         letters_LCD_two = letters_LCD_two >> 8;
466         letters_LCD_two[127:120] = letter_two;
467     end
468     default: Y_D = BREAK;
469 endcase
470 end
471
472 /* newbreak == 1 since ore == newbreak ^ newdot
473 and newdot is false */
474 0: case (v)
475     0:
476         begin
477             Y_D = BREAK;
478             case (y_Q)
479                 E: letter_one = LE;
480                 T: letter_one = LT;
481                 I: letter_one = LI;
482                 A: letter_one = LA;
483                 N: letter_one = LN;
484                 M: letter_one = LM;

```

```

485         S: letter_one = LS;
486         U: letter_one = LU;
487         R: letter_one = LR;
488         W: letter_one = LW;
489         D: letter_one = LD;
490         K: letter_one = LK;
491         G: letter_one = LG;
492         O: letter_one = LO;
493         H: letter_one = LH;
494         V: letter_one = LV;
495         F: letter_one = LF;
496         L: letter_one = LL;
497         P: letter_one = LP;
498         J: letter_one = LJ;
499         B: letter_one = LB;
500         X: letter_one = LX;
501         C: letter_one = LC;
502         Y: letter_one = LY;
503         Z: letter_one = LZ;
504         Q: letter_one = LQ;
505
506     endcase
507     y_Q = Y_D;
508     //Shift letter's value into the register
509     letters_LCD_one = letters_LCD_one >> 8;
510     letters_LCD_one[127:120] = letter_one;
511     #1000; //prevent completion of the block
512 end
513
514 1:
515     begin
516         letter_one = SPACE;
517         letter_two = SPACE;
518         Y_D = BREAK;
519         y_Q = Y_D;
520
521         //Shift letter's value into the register
522         letters_LCD_one = letters_LCD_one >> 8;

```

```

523         letters_LCD_one[127:120] = letter_one;
524         letters_LCD_two = letters_LCD_two >> 8;
525         letters_LCD_two[127:120] = letter_two;
526         #1000; //prevent completion of the block
527     end
528 endcase
529 endcase
530 end // state_table
531
532 //Display the two rows
533 LCD disp (CLOCK_50, KEY[1:0], GPIO_0, GPIO_1,
           LCD_ON, LCD_BLON, LCD_RW, LCD_EN,
534 LCD_RS, PS2_DAT, PS2_CLK, LCD_DATA,
           letters_LCD_one, letters_LCD_two);
535 endmodule

```

The module named *LCD* which is referred to in the code above is a slight modification of a previously used 'lcdlab' which can be found at <http://www.johnloomis.org/digitallab/lcdlab/lcdlab3/lcdlab3.html>

The modification simply allowed the module to display the registries that stored the values of both rows of the LCD.

A portion of the FSM used for the transition of characters. There are 27 states: one for each letter and one for the **BREAK** state.

