Programming the RC2014 Front Panel

Part 2: Z80 ASSEMBLY

Bruce E. Hall, W8BH

In the previous article, I described how to program the front panel LEDs, switches, and character LCD using Microsoft BASIC. In this article, Z80 assembly language is used to animate the display.

WHICH ASSEMBLER?

I use the Telemark Assembler (TASM) to compile the code on my Windows PC, then transfer the object file to my Z80 machine over X-modem. If you use a different assembler you may need to adjust the syntax. But the concepts will be the same. Here is my process:

```
TASM -80 -g3 demo.asm   (create DEMO.OBJ via a Windows DOS box)
XM R DEMO.COM           (receive the file as DEMO.COM file in CP/M)
```

For an introduction to using TASM and Z80 assembly, see Part 2 of my Assembly Language Programming for the RC2014 article.

SENDING CHARACTERS

With BASIC, we can show a character on screen with a single OUT instruction. It's a bit more complicated in Z80 assembly. Why? Because at assembly speeds, the computer is faster than the LCD controller. After every OUT instruction, we must check to see if the LCD controller is ready to accept another character. It signals its readiness by dropping bit7 on the command port. So, after each out, we repeatedly IN the command port, looking at bit7. A convenient way to do this is to rotate the byte left (with bit7 going into the carry bit) and looping until the carry is clear. This wait loop is highlighted in yellow below.

```
SendChar:
    OUT (dataPort),A          ; send a character to LCD
    IN A,(cmdPort)            ; check LCD status (bit 7 = busy)
    RLCA                      ; move status bit to carry
    JR  C,$-3                 ; wait until ready
    RET
```

In the above code, “$-3” is shorthand for “the address 3 bytes before the current instruction pointer”. In other words, the address of the IN instruction. Sending a command to the LCD controller is the same, except that the byte is sent to the controller’s command port instead of the data port.

Filling the screen is simply a matter of clearing the screen, then sending the character 80 times. We can make it more interesting by getting the character from the command tail. In other words, type “LCDFILL 6” will fill the screen with 6’s. In CP/M 2.2, the first character of the command tail is located at absolute address $0082, so a load from that address is all we need.

```
Start:
    CALL ClearScreen
    LD   A,($0082)            ; get 1st char of command tail
    LD   C,A                  ; save char in C
    LD   B,80                 ; count 80 characters
j0:
    LD   A,C                  ; retrieve char
    CALL SendChar             ; send it to the LCD
    DJNZ j0                   ; repeat until all sent
    RET
```

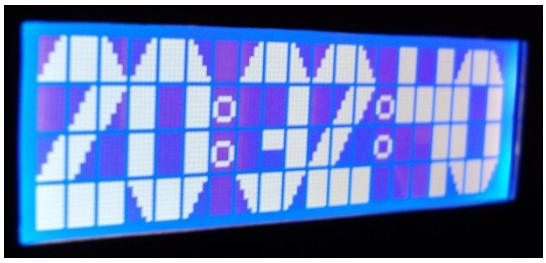
The code for LCDFILL.ASM can be downloaded from my GitHub page.

DISPLAYING THE COMMAND TAIL

Instead of just sending the first character in the command tail, we can send the entire string. This turns out to be useful program, giving you the ability to post messages from the operating system environment. In CP/M, the first byte of the command tail holds the length of the string. Notice the highlighted similarities to the program above. The complete code for LCD.ASM is on GitHub. Invoking it with no text clears the screen.

```
Start:
    CALL ClearScreen
    LD HL,$0080               ; point to command tail
    LD B,(HL)                 ; get tail length
    INC HL                    ; ignore 1st char (space)
    DEC B
j0:
    INC HL                    ; point to next char
    LD A,(HL)                 ; get it
    CALL SendChar             ; & show it on LCD
    DJNZ j0                   ; loop for all chars
    RET
```

BIG CLOCK



Now that looks like 20x4 LCD fun. But how do you program it?

Stare at the image for a while, and you'll see that each digit is rendered in a 3x4 block of characters. And each character is made up of just a few symbols: a solid block, some triangles, and some half-height blocks.

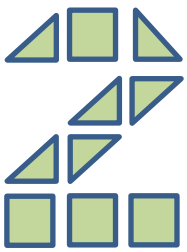
In part I used BASIC to create custom characters. Let's do the same in Z80 assembly. You can emulate the big digits with 8 different symbols: a lower-right triangle, lower-left triangle, upper-right triangle, upper-left triangle, upper horizontal bar, lower horizontal bar, and colons. Here they are:

```
BD_ICONS  .DB $01,$03,$03,$07,$07,$0F,$0F,$1F  ; lower-rt triangle
          .DB $10,$18,$18,$1C,$1C,$1E,$1E,$1F  ; lower-lf triangle
          .DB $1F,$0F,$0F,$07,$07,$03,$03,$01  ; upper-rt triangle
          .DB $1F,$1E,$1E,$1C,$1C,$18,$18,$10  ; upper-lf triangle
          .DB $00,$00,$00,$00,$1F,$1F,$1F,$1F  ; lower horiz bar
          .DB $1F,$1F,$1F,$1F,$00,$00,$00,$00  ; upper horiz bar
          .DB $00,$0e,$11,$11,$11,$0e,$00,$00  ; open dot
          .DB $00,$0E,$1f,$1f,$1f,$0e,$00,$00  ; closed dot
```

Each digit is made up of a 3x4 grid of these symbols. I created a list with 10 members for the 10 digits. Each member is a list of the 12 symbols needed to create the digit.

```
BIGDIGITS .DB $00,$FF,$01,$FF,$20,$FF,$FF,$20,$FF,$02,$FF,$03   ; #0
          .DB $00,$FF,$20,$20,$FF,$20,$20,$FF,$20,$20,$FF,$20   ; #1
          .DB $00,$FF,$01,$20,$00,$03,$00,$03,$20,$FF,$FF,$FF   ; #2
          .DB $00,$FF,$01,$20,$20,$FF,$20,$05,$FF,$02,$FF,$03   ; #3
          .DB $FF,$20,$FF,$FF,$FF,$FF,$20,$20,$FF,$20,$20,$FF   ; #4
          .DB $FF,$FF,$FF,$FF,$04,$04,$20,$20,$FF,$FF,$FF,$03   ; #5
          .DB $00,$FF,$01,$FF,$20,$20,$FF,$05,$01,$02,$FF,$03   ; #6
          .DB $FF,$FF,$FF,$20,$20,$FF,$20,$20,$FF,$20,$20,$FF   ; #7
          .DB $00,$FF,$01,$FF,$20,$FF,$FF,$05,$FF,$02,$FF,$03   ; #8
          .DB $00,$FF,$01,$02,$04,$FF,$20,$20,$FF,$20,$20,$FF   ; #9
```

$00 indicates the first symbol (a lower-right triangle), $01 indicates the second (a lower-left triangle), and so on. $20 is a blank. The 12 symbols in each digit are ordered from top-left to lower-right.



The numeral '2' begins with a lower-right triangle ($00) in the top left corner, followed by a solid block ($FF) and a lower-left triangle ($01). The complete sequence is $00, $06, $01, $20, $00, $03, $00, $03, $20, $06, $06, $06

To display a digit, take its list of 12 symbols and display them in a 3x4 matrix:

```
; Call with A=digit value and C=column offset
;
ShowBigDigit:
    LD   E,12               ; DE = size of each digit defn
    LD   D,0
    LD   HL,BIGDIGITS       ; point to digit definitions
    OR   A                  ; is A=0?
    JR   Z,sbd2             ; for digit zero, we are done
    LD   B,A
sbd1:
    ADD  HL,DE              ; point to next digit defn
    DJNZ sbd1               ; until desired digit reached
sbd2:                       ; now HL = HL + (12*digit)
    LD   DE,digitXY         ; point to icon location array
    LD   B,12               ; each digit is 3x4=12 icons big
sbd3:
    LD   A,(DE)             ; get icon position
    ADD  A,C                ; add in column offset
    ADD  A,SET_CURSOR       ; make it a command
    CALL SendCommand        ; set icon position
    LD   A,(HL)             ; get icon to display
    CALL SendChar           ; display it
    INC  DE                 ; goto next position
    INC  HL                 ; goto next icon
    DJNZ sbd3               ; loop for all 12 positions
    RET
```

The first part of this routine points the HL register pair to BIGDIGITS and adds 12 to it repeatedly (loop sbd1) until the desired digit definition is reached. For example, to display a 5, it will add 12 to HL 5 times, so that at label sbd2, register pair HL has advanced 60 bytes to point at the definition of big digit 5.

The last part of the routine, starting at label sbd3, copies all 12 icons from the digit definition to the display. Register pair DE points to an array that specifies the cursor location where each of the 12 icons should be placed: the first icon goes to cursor position 0; the second to cursor position 1, etc. The starting row addresses on this LCD controller are not sequential: adding a character to the last column of the top row does not advance the cursor to the start of the next, as expected, but to the 3rd row. Using an array of display addresses helps navigate this peculiar layout.

20x4 Display	Starting Address
Row 0 (top)	$00
Row 1	$40
Row 2	$14
Row 3 (bottom)	$54

The remainder of the code handles advancing the time by hours, minutes, and seconds, and displaying those values in big digits. See CLOCK4.ASM and CLOCK6.ASM for details.

ANIMATED BAR GRAPHS

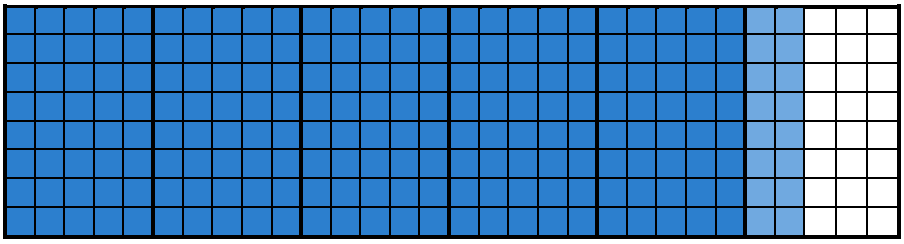
In Part 1, we used BASIC to create a horizontal bar graph. Now let's animate the graph. Instead of changing the length of the bar in one jump, say from 33 to 47, we draw the bar incrementally with a length of 33, then 34, then 35, … finally stopping at 47.

In BASIC, we used integer division and the modulo (remainder) function for determining the number of full and partial blocks to display. Since those functions are not available in the Z80 instruction set, we must use serial subtraction instead:

```
DrawHBar:
    LD   A,C                ; look at current value
    SUB  5                  ; remove 5 from it
    LD   C,A                ; save result
    JR   C,dhb0             ; skip if less than 5
    LD   A,$FF
    CALL SendChar           ; draw full 5 bars
    JR   DrawHBar
dhb0:                       ; get here when remainder <5
    RET  Z                  ; if 0, no need to print more
    ADD  A,5                ; undo last subract
    CALL SendChar           ; print it as 1-4 bars
    RET
```

The routine starts with the length of the bar in register C. The first half of the routine forms a loop, where 5 is subtracted and a full block is printed until the register value goes negative (carry bit set).

For example, a bar graph of length 27 will be 5 full blocks (5*5=25), followed by a character containing the last 2 bars:



Bar of 27 units =

5 full blocks +
1 partially filled block of 2 bars

For this example, the code would loop 5 times, each time subtracting 5 and printing a full block each time. At that point, the remainder in C would be 2. When the sixth subtraction is done, the value goes negative (2-5=-3) and the loop exits. Finally, the remainder of 2 is restored by adding back 5. The remainder is used to print a partial block containing 2 bars.

Animating the graph is done by drawing the bar and adjusting its length by one (either up or down) until reaching the new value. Before calling this routine, a value of +1 or -1 is placed in variable SIGN to specify whether the length should increase or decrease.

```
AnimateHBar:
    LD   A,ROW              ; set cursor location
    LD   B,COL
    CALL GotoRC
    LD   C,D
```

```
CALL DrawHBar           ; draw the bar
LD   BC,SPEED
CALL DELAY_BC           ; animation delay
LD   A,(sign)
ADD  A,D                ; incr/decr width by 1 bar
LD   D,A                ; update bar width
LD   A,(newValue)
CP   D                  ; compare with new value
JR   NZ,AnimateHBar     ; loop until NV reached
RET
```

See HBAR1.ASM for a single animated bar, and HBAR.ASM for an animated graph.

Vertical bar graphs are done in the same manner, using an icon set of incrementally taller bars. The logic is the same. See VBAR.ASM for an animated vertical bar graph.

ANIMATED SINE WAVE

It's fun to look at and easy to program. Animation is accomplished by incrementing the starting phase of sin(x) each time the graph is displayed. Instead of computing sin(x), which requires a lot of 8-bit computing power, we use a small lookup table instead:

```
sine      .DB 16,22,27,30,31,29,25,19,13,7,3,1,2,5,10

GetSine:
    LD   D,0
    LD   E,A                ; DE = A
    LD   HL,sine            ; point to LUT
    ADD  HL,DE              ; point to result
    LD   A,(HL)             ; get result in A
    RET
```

This table gives us a value of sin(x) every 24 degrees. The values are linearly translated (Y=15*sin(24x)+16) to return an integer result in the range of 0-31. GetSine only needs to return the value in the sine table indexed by A. See SINE.ASM for the full program.

PUTTING IT ALL TOGETHER

I gathered up a dozen of these front-panel demos and put them in a single program called FPDEMO.ASM. The code leaves room for four more demos. Have fun creating your own demos and adding them to the program.

Bruce.

This text of these articles and the coding examples are my own, created for the pure enjoyment of it. I have not used AI or any AI-generated material.

Last Updated: 30 Aug 2026