14 Testing the Graphic LCD Panel

Although Steve Ciarcia still names "Solder" as his favorite programming language, even he admits most projects nowadays are useless without some programming. Now, with the molten metal part of the Graphic LCD Interface finished, it's time for the bit twiddling!

I'll describe the test code that exercises my panels and explain some tricks and techniques. Although Micro-C predominates in this chapter, you'll surely want assembler code once you understand how your panel works. If pushing dots around doesn't make a good case for hand tweaked optimization, well, what does?

But first, let's check out your wiring. Because this project has lots of fairly tricky hardware, I'm going to spend more time than usual on bringup testing. With any luck at all, your circuitry and LCD panel will work perfectly the first time. If they don't, these hints may come in handy.

Testing...

The `GraphLCD` files include all the Graphic LCD Interface test code. You can use the `BIN` file directly or, if you must make changes for your panel, recompile the Micro-C source into an Intel `HEX` file and convert it into binary. Copy the result to a diskette with the appropriate boot sector loader from Chapter 1, connect a serial cable, pop the floppy into your target system, hit the Reset button, and select the appropriate option when `GraphLCD` squirts its menu through the serial port to the comm program running on your host PC.

If you prefer Borland or Microsoft C and have Paradigm's `Locate` (or a similar utility) available, you can translate the Micro-C code into their dialects fairly easily, using the hints in Chapter 11. In any event, don't despair: the hardware neither knows nor cares which C you use.

You'll recall that the CPU's view of the Graphic LCD Interface includes one write-only output port, two 82C54 timer channels, and a 32 KB block of RAM. `GraphLCD`'s first four tests wiggle the control port bits, give you manual control over the 82C54, and read and write the **`LCD Refresh RAM`** in loops.

These tests work correctly even without an LCD panel, allowing you to start wiring and testing while waiting for your new panel to arrive. If you *do* connect a panel, remember that it won't display anything without the correct sync signals and jumper settings. I *strongly* recommend delayed gratification: keep that panel in the box, safely wrapped in its antistatic bag, until your hardware passes its tests!

First, verify that all eight bits in U51, the **LCD Control Latch**, go ON and OFF in the proper order. As it turns out, I managed to wire the port backwards and this simple test prevented some serious headscratching.

Next, set the 82C54 row and frame counters. With no LCD panel attached (right?), you can specify three clocks per row and nine clocks per frame to simplify your scope display. Each row must have at least two clocks and, of course, the number of clocks in each frame must be a multiple of the row clock count.

Scope the **LCD Address Counter**, U43 and U44, to verify its inputs and outputs. The LS590 latch outputs go active (well, they *should* go active) only during the low half of the **+Dot Clock** cycle. The **Frame Sync** pulse resets the counters with a signal from U56B, so check that the count begins with zero at the right time. Refer back to Figure 1 in Chapter 13 for guidance.

The RAM read and write tests toggle printer port bits at key points in the loop to provide oscilloscope triggering. The read test uses a single `REP LODSB` to touch every **LCD Refresh RAM** byte and produces one sync pulse at the start of that lengthy instruction. The write test touches each RAM byte in a C loop and, as this requires much more time, issues a sync pulse before every RAM access. In either case, U56A should stall **+Dot Clock** in the high half of its cycle at the end of each **-SMemR** or **-SMemW** pulse.

The fifth test generates and writes a 32 KB pseudorandom pattern into the **LCD Refresh RAM**, reads the pattern back, and verifies that the bytes return from their journey unchanged. While this isn't an exhaustive RAM test, it will reveal obvious wiring errors, short circuits, and dead chips. If you don't have the equipment required for the other tests, just run this one and hope it works. Should it fail, you now have a problem that merits investigation and, perhaps, some motivation to acquire that collection of test equipment you've always wanted.

The RAM test may reveal intermittent data errors if your **LCD Refresh RAM** or buffers run too slowly for the ISA bus accesses. Remember that the RAM access must occur in slightly less than half of a **Dot Clock** cycle, about 240 ns. My numbers indicate that a 120 ns RAM works OK with my hardware, but your mileage may differ. In any event, don't install your LCD panel until the RAM works perfectly: it's hard to find other bugs when you can't trust your test data.

Once your bits fly in formation, check and recheck the LCD power supply voltages. If you're using the LM337 circuit shown in Chapter 13, remember that you can destroy your panel with one twist of R65. Set the LCD bias supply voltage to match the value in your panel's data sheet. If your panel requires a contrast voltage, also adjust R66 to midrange.

Next, build a cable that mates the Graphic LCD Interface's 2×13 header to your panel. If you can find a connector for your panel, great. Otherwise, just solder the appropriate ribbon cable wires directly to the panel's header, as you saw in Photos 1 and 2 in Chapter 12, and be done with it. I generally tape the cable to the back of the panel while testing the hardware. You can surely come up with a better, more permanent arrangement for your final installation.

If you do find an LCD backlight inverter, wire it to the panel through a separate cable. Don't succumb to the temptation of running that 1 kV, high frequency, AC power through the same cable as your precious sync and data signals. Just say "No!"

The final step requires nothing more than a simple ohmmeter continuity check: verify the connections from the Graphic LCD Interface circuitry to the LCD panel. You have only one last chance to get the power supply voltages on the right pins: don't blow it now.

Refer back to Photo 1 in Chapter 12, where you'll see that the Optrex DMF651 640×200 panel sports a helpful, silk screened legend on the upper left corner of the circuit board:

UP ↑

All of the other panels in my collection assume you can derive this key datum from their documentation. Alas, the doc you get with the panel sometimes won't give even the barest hint as to which end is up. In that case, the remaining tests will not only help you debug the hardware, but get the panel bolted down properly.

OK, plug it in and let's do some dots…

...Testing! One...

Perhaps the first thing you asked about a new graphic LCD panel was "How big is it?" From the number of dot rows and columns, you can calculate the required size of the **`LCD Refresh RAM`** in bytes. The DMF651 is a 640×200 panel that displays 128,000 bits from 16,000 bytes (not quite 16 KB) of data.

The Graphic LCD Interface measures LCD panels somewhat differently. Rather than rows and columns, both you and the hardware must know the number of **`Dot Clock`**s in each row, the total number of **`Dot Clock`**s in a complete frame, and the number of data bits transferred on each **`Dot Clock`**. As you saw in Chapter 13, though, there may be only the slightest relation between a panel's number of dots and the signals that control it.

For example, the DMF651 accepts four bits on each **Dot Clock** and, thus, transfers each 640-dot row in 160 clocks. The panel has 200 row drivers, one for each physical row, and a frame requires 32,000 **Dot Clock** cycles. The dots occupy one nybble in each of the first 32,000 **LCD Refresh RAM** bytes.

The test code in Listing 1 initializes the Graphic LCD Interface hardware for a DMF651 panel, then writes the test pattern into the **LCD Refresh RAM**. Listing 2 defines the **LCD Control Port** bit patterns used in the code. The code for

Listing 1

The Optrex DMF651 640x200 LCD panel has a comparatively simple layout: a single frame requires 200 sets of 160 Dot Clocks, each transferring four bits. This code fragment shows how to load the 82C54 timers and set the LCD Control Port. The SetData routine puts four data bits in the low nybble and creates the right bits for the high nybble to produce the desired blinking effect.

```
outpw(GLCD_CTLS,GLCD_OFF);                          /* disable clocking      */
LoadTimer(1,0x04,160,I8254_BASE);                   /* clocks per row        */
LoadTimer(2,0x04,32000,I8254_BASE);                 /* total clocks/frame    */
outpw(GLCD_CTLS,GLCD_ON + GLCD_BLMED);              /* start and set blink   */
for (Row = 0; Row < 200; Row++) {
   RAMAddr = Row * 160;
   for (Col = 0; Col < 160; Col++) {
      poke(GLCD_SEGMENT,RAMAddr+Col,0x00);          /* clear row             */
   }
   SetData(RAMAddr+0,BLINK_OFF,Row >> 4);           /* show row number       */
   SetData(RAMAddr+1,BLINK_OFF,Row);
   SetData(RAMAddr+2,BLINK_FG,0x09);                /* dividing line         */
   SetData(RAMAddr+3,BLINK_OFF,RAMAddr >> 12);      /* show RAM address      */
   SetData(RAMAddr+4,BLINK_OFF,RAMAddr >> 8);
   SetData(RAMAddr+5,BLINK_OFF,RAMAddr >> 4);
   SetData(RAMAddr+6,BLINK_OFF,RAMAddr);
   SetData(RAMAddr+7,BLINK_FG,0x09);                /* dividing line         */
   SetData(RAMAddr+8+(Row >> 2),BLINK_BG,0x08 >> (Row & 0x03));/* diagonal */
}
```

Listing 2

These definitions combine useful LCD Control Port bit patterns into easily remembered groups. You should change the GLCD_OFF and GLCD_ON values to match your panel.

```
#define GLCD_ENCTR      0x0001              /* enable addr counters          */
#define GLCD_DISMUX     0x0002              /* 1 disabled output mux         */
#define GLCD_ENDISP     0x0004              /* enable LCD (if used)          */

#define GLCD_BLSLOW     0x0050              /* slowest blinking              */
#define GLCD_BLMED      0x0040              /* moderate blinking             */
#define GLCD_BLFAST     0x0030              /* fastest blinking              */
#define GLCD_BLZERO     0x0000              /* output logic zero             */
#define GLCD_BLCLOCK    0x0060              /* output LCD clock              */
#define GLCD_BLONE      0x0070              /* output logic one              */

#define GLCD_OFF        (GLCD_DISMUX | GLCD_BLZERO)
#define GLCD_ON         (GLCD_ENCTR | GLCD_ENDISP)
```

this chapter also produces several trace outputs which I've removed from these listings to save space; refer to the complete source files for all the details.

Recall that both 82C54 timer channels must begin counting on the same **Dot Clock** cycle. The first line disables the counters by lowering their **Gate** inputs. After loading the row and frame lengths, the code enables the 82C54 and sets the blink rate. The Graphic LCD Interface immediately begins sending data, **Dot Clock**s, and sync pulses to the panel, even though the **LCD Refresh RAM** contains no useful data.

Although it may seem strange to start the sync signals before clearing the RAM, that ensures you'll see *something* on the panel, even if the memory interface isn't working quite right. Should your panel display random bits instead of the test pattern, you know where to start looking for the problem. If the pattern changes, then you can assume the program tried and failed to write the test pattern.

The test loop then iterates 200 times, once for each row on the LCD panel. The rows occupy contiguous regions in RAM and, because each row uses 160 bytes of RAM, their starting addresses are 160 (decimal) times the row number: 0000,

Listing 3

Panels such as the Optrex DMF651 accept four bits per Dot Clock cycle, so the Graphic LCD Interface can produce blinking by switching between the two nybbles in each byte several times per second. This routine puts the data bits in the low nybble, then sets the high nybble to produce the desired blink pattern.

```
SetData(Addr,Blink,Data)
WORD Addr;
int  Blink;
WORD Data;
{
   Data &= 0x000F;                    /* force high nybble off            */
   switch (Blink) {
   default :
      printf("Invalid blink mode = %u\n",Blink);
   case BLINK_OFF :
      Data |= Data << 4;
      break;
   case BLINK_FG :
      break;
   case BLINK_BG :
      Data |= 0x00F0;
      break;
   case BLINK_INV :
      Data |= (~Data) << 4;
      break;
   }
   poke(GLCD_SEGMENT,Addr,Data);
}
```

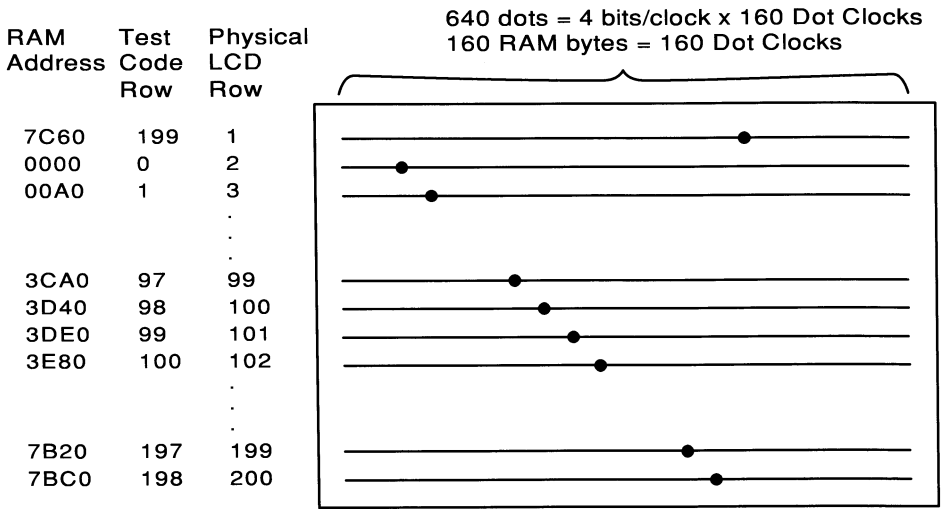
00A0, 0140, and so on. I set `RAMAddr` to the beginning of the current row at the start of each iteration to simplify the rest of the code in the loop.

The test routines take advantage of the LCD's high dot density to display the row number and RAM address in binary format in the first few dozen dots of each row. If your eyes work like mine, expect to whip out a magnifying glass to read the dot patterns that give each line an unambiguous identifier. A diagonal line provides an easy visual check that all the rows appear in the right order.

Remember the sound of one hand clapping in Chapter 13?

Figure 1 sketches the row numbers, **`LCD Refresh RAM`** addresses, and the diagonal line for a DMF651. The test code numbers the top row 199 because it's the *last* one in the buffer… but the panel displays those dots on the *first* row because they *precede* the **`Frame Sync`** pulse. That puts one lonely dot off on the right of the top row, aligned above the end of the diagonal line.

Figure 1
The Optrex DMF651 has a fairly simple layout. The top row of dots appears to be out of sequence, because the test code numbers the visible rows starting with Row 0 at address 0000. The Graphic LCD Interface hardware produces a Frame Sync pulse when it resets the LCD Address Counters to 0000 and the data immediately before the pulse comes from the highest RAM addresses. The LCD panel displays the bits arriving before the Frame Sync pulse on the top row, which puts the test code's Row 199 on top.



Listing 3 shows the `SetData` routine that writes bit patterns into RAM. Although I defined only four blinking patterns, you can certainly come up with some truly bizarre effects. The initial **`LCD Control Port`** setup determines the blinking rate, as shown in Listings 1 and 2.

I started with the DMF651, because it's about as simple a panel as you're likely to encounter: the physical dot layout even matches the electrical sync signal pattern! Sad to say, such simplicity forms the exception rather than the rule. Many other panels can give you a nasty case of brain burn…

The Toshiba TLY-365-121, also a 640×200 panel, uses 320 **`Dot Clock`**s that transfer 1280 bits on each of 100 rows. The firmware sees it as a 1280×100 array and maps the dots accordingly. Listing 4 shows the test pattern code, with a few key differences in the 82C54 setup and `RAMAddr` calculation.

The loop still iterates 200 times, once for each visible dot row, but finding the row starting address becomes somewhat more complex. The first 160 bytes appear on one row, with the remaining 160 bytes starting 100 rows further down the panel. Row 99 is on top, with Row 199 in the middle of the panel. Ah, yes, do you remember row counts use *decimal* notation?

To prove I'm not making this up, Photo 1 shows the actual test pattern on the panel. If you can't see the dots clearly, Figure 2 sketches the layout.

Listing 4
The Toshiba TLY-365-121 has 1280 dots in each of 100 logical rows, with 320 Dot Clocks on each row. Compare the LoadTimer() parameters and SetData patterns with Listing 1 to see the effect of the double-length lines.

```
outpw(GLCD_CTLS,GLCD_OFF);                          /* disable clocking      */
LoadTimer(1,0x04,320,I8254_BASE);                   /* clocks per row        */
LoadTimer(2,0x04,32000,I8254_BASE);                 /* total clocks/frame    */
outpw(GLCD_CTLS,GLCD_ON + GLCD_BLMED);              /* start and set blinking*/
for (Row = 0; Row < 200; Row++) {
   RAMAddr = 320 * (Row % 100) + ((Row > 99) ? 160 : 0);
   for (Col = 0; Col < 160; Col++) {
      poke(GLCD_SEGMENT,RAMAddr+Col,0x00);          /* clear row             */
   }
   SetData(RAMAddr+0,BLINK_OFF,Row >> 4);           /* show row number       */
   SetData(RAMAddr+1,BLINK_OFF,Row);
   SetData(RAMAddr+2,BLINK_FG,0x09);                /* divider               */
   SetData(RAMAddr+3,BLINK_OFF,RAMAddr >> 12);      /* show RAM address      */
   SetData(RAMAddr+4,BLINK_OFF,RAMAddr >> 8);
   SetData(RAMAddr+5,BLINK_OFF,RAMAddr >> 4);
   SetData(RAMAddr+6,BLINK_OFF,RAMAddr);
   SetData(RAMAddr+7,BLINK_INV,0x09);               /* divider               */
   SetData(RAMAddr+8+(Row >> 2),BLINK_BG,0x08 >> (Row & 0x03));/* diagonal */
}
```

Photo 1
The test pattern on this Toshiba TLY-365-121 640x200 LCD panel is based on Figure 2. The first several dozen dots in each row display the test code's row number (0 through 199) in binary, the row's RAM address, and a diagonal line. The code also inserts blinking separator patterns between the data values (which, of course, appear solid in this photograph).

You may not be able to see the two "stray" dots on the top and middle rows of the panel. Trust me: they're visible on the original slide and you'll see them on your own panel!

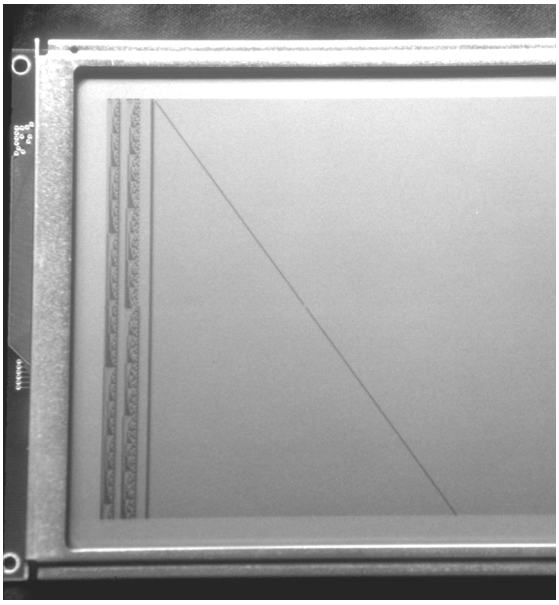
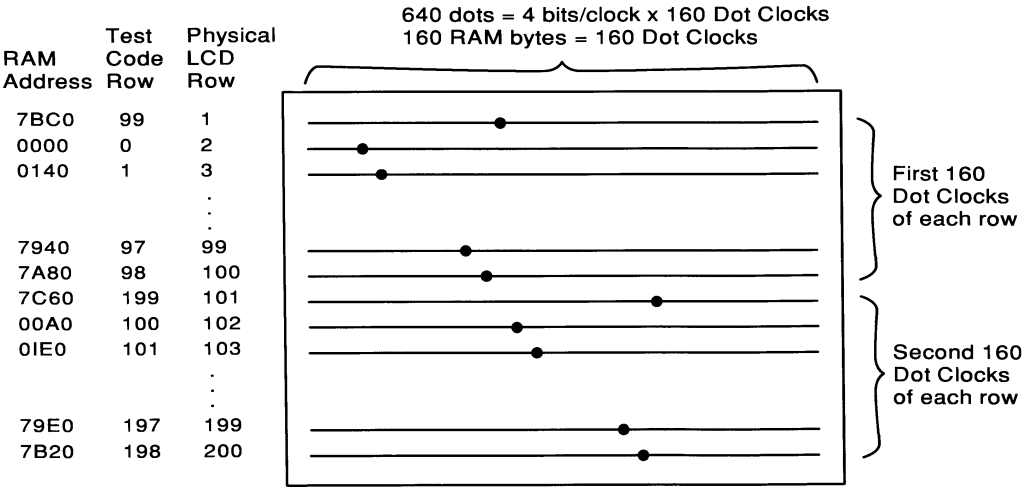


Figure 2
The Toshiba TLY-365-121 splits each logical row of 320 Dot Clocks in two: the first 160 clocks transfer 640 dots to the upper half of the panel, while the second 640 dots appear in the lower half. Therefore, the 320 Dot Clocks before the Frame Sync pulse appear on physical rows 1 and 101. Photo 1 shows this test pattern in real life.



... Two...

As I mentioned in the previous chapter, LCD designers have recourse to only two methods that provide the increased bandwidth required by 400-line panels: send more dots per clock or stuff more clocks into each frame. The Graphic LCD Interface can handle either method, provided that you set up the right bits and jumpers before turning the power on.

The Matsushita EDM LG64AA44D accepts eight data bits on each **Dot Clock** cycle. The **LCD Control Port** value sends a constant zero to the **LCD Data Multiplexer**, forcing it to gate only the low nybble to the panel. The high nybble connects directly to the panel, presenting the entire **LCD Data Latch** on each **Dot Clock** cycle. Listing 5 shows the test code.

As the loop iterates through all 400 visible lines, `RAMAddr` goes through the same set of 200 addresses twice. With no spare bits for blinking, the `StoreNybble`

Listing 5

The Matsushita EDM LG64AA44D panel has 640x400 physical dots, but the firmware sees it as two 640x200 panels. The test code loops through all 400 rows and sets RAMAddr to the start of each one. The value of Align selects the appropriate nybble and the StoreNybble function inserts the data bits. Because the row number exceeds 255 for this panel, the firmware displays it with 16 dots.

```
outpw(GLCD_CTLS,GLCD_OFF);                          /* disable clocking      */
LoadTimer(1,0x04,160,I8254_BASE);                   /* clocks per row        */
LoadTimer(2,0x04,32000,I8254_BASE);                 /* total clocks/frame    */
outpw(GLCD_CTLS,GLCD_ON | GLCD_BLZERO);             /* get bits 0-3 from mux */

for (Row = 0; Row < 400; Row++) {
   RAMAddr = 160 * (Row % 200);                     /* buffer address        */
   Align = (Row >= 200) ? 4 : 0;                    /* high/low bit align    */

   for (Col = 0; Col < 160; Col++) {
      StoreNybble(RAMAddr+Col,Align,0);             /* clear our nybble      */
   }
}
for (Row = 0; Row < 400; Row++) {
   RAMAddr = 160 * (Row % 200);                     /* buffer address        */
   Align= (Row >= 200) ? 4 : 0;                     /* high/low bit align    */

   StoreNybble(RAMAddr+0,Align,Row >> 12);
   StoreNybble(RAMAddr+1,Align,Row >> 8);
   StoreNybble(RAMAddr+2,Align,Row >> 4);
   StoreNybble(RAMAddr+3,Align,Row);
   StoreNybble(RAMAddr+4,Align,0x08);               /* divider lines         */
   StoreNybble(RAMAddr+5,Align,RAMAddr >> 12);
   StoreNybble(RAMAddr+6,Align,RAMAddr >> 8);
   StoreNybble(RAMAddr+7,Align,RAMAddr >> 4);
   StoreNybble(RAMAddr+8,Align,RAMAddr);
   StoreNybble(RAMAddr+9,Align,0x06);               /* divider lines         */
   StoreNybble(RAMAddr+10+(Row >> 2),Align,0x08 >> (Row & 0x03));
}
```

function shown in Listing 6 inserts four data bits in the proper half of the byte, as dictated by the `Align` variable.

The Sharp LM64015T panel uses a double-speed 240 ns **`Dot Clock`**, selected by jumper JP12. The **`LCD Refresh RAM`** runs at the same 480 ns rate as the other panels, so the code in Listing 7 routes **`Dot Clock`** to the **`LCD Data Multiplexer`**, switching it between the two **`LCD Data Latch`** nybbles every 240 ns. The high nybble goes out in the first half cycle, which, conveniently enough, lays the dots out left to right on the panel.

Because only the LM64015T sees the 240 ns **`2x Dot Clock`**, the rest of the Graphic LCD Interface thinks the panel has just 160 ordinary rows with 480 ns **`Dot Clock`** pulses apiece and 32000 clocks per frame. The timing remains the same as the DMF651, but you must disable blinking because all the bits in each byte contribute to the normal display. JP10 and JP11 select the double-speed sync signals that stay active for only half of the normal **`Dot Clock`** cycle.

Unlike the other panels, the LM64015T turns a 1 bit into a transparent dot, as the panel starts off opaque and the backlight shines through the ON bits. I find this disconcerting, but you can easily display dots either way. The test pattern uses dark data dots to show the technique: note that the lighted dots each have a dark border.

The LM641481 640×480 panel resembles its 400-line ancestor, although, as you saw in Chapter 13, you must enlarge the **`LCD Refresh RAM`** to hold all the new dots. `GraphLCD` includes a test pattern for this panel and others like it, assuming that you build the hardware for it. Running with a 32 KB RAM will duplicate some of the dots, because a 15-bit address repeats the RAM contents on the higher rows. Refer back to Chapter 13 for the hardware solution.

Listing 6

The LG64AA44D accepts twice as much data per frame through its 8-bit interface as 200-line panels. This code fetches a byte from the buffer, clears the selected nybble, inserts the new bits, and then stores the result back. Because ISA memory runs quite slowly, you should use a duplicate buffer in system RAM to eliminate the extra read.

```
StoreNybble(Addr,Align,Data)
WORD Addr;
int  Align;
WORD Data;
{
WORD OldData;

   OldData = peek(GLCD_SEGMENT,Addr) & (0xf0 >> Align);
   Data = ((Data & 0x0f) << Align) | OldData;
   poke(GLCD_SEGMENT,Addr,Data);
}
```

Listing 7
The Sharp LM64015T runs at twice the speed of the other panels, accepting a nybble on each half cycle of the 480 ns Dot Clock. The LCD Data Multiplexer switches between the two nybbles of each Refresh RAM byte. A jumper on the Graphic LCD Interface routes the double-speed 2x Dot Clock to the panel. Note the inverted data: a 1 bit sent to the panel appears transparent rather than opaque.

```
outpw(GLCD_CTLS,GLCD_OFF);                          /* disable clocking      */
LoadTimer(1,0x04,160,I8254_BASE);                   /* Dot Clocks/row        */
LoadTimer(2,0x04,32000,I8254_BASE);                 /* total clocks/frame    */
outpw(GLCD_CTLS,GLCD_ON | GLCD_BLCLOCK);            /* alternate at clk rate */
for (Row = 0; Row < 400; Row++) {
   RAMAddr = 160 * (Row % 200) + ((Row > 199) ? 80 : 0);
   for (Col = 0; Col < 80; Col++) {
      poke(GLCD_SEGMENT,RAMAddr+Col,0xff);          /* 1 = clear             */
   }
}
for (Row = 0; Row < 400; Row++) {
   RAMAddr = 160 * (Row % 200) + ((Row > 199) ? 80 : 0);
   poke(GLCD_SEGMENT,RAMAddr+0,~(Row >> 8));
   poke(GLCD_SEGMENT,RAMAddr+1,~(Row & 0x00FF));
   poke(GLCD_SEGMENT,RAMAddr+2,~0x99);              /* divider lines         */
   poke(GLCD_SEGMENT,RAMAddr+3,~(RAMAddr >> 8));
   poke(GLCD_SEGMENT,RAMAddr+4,~(RAMAddr & 0x00FF));
   poke(GLCD_SEGMENT,RAMAddr+5,~0x99);              /* divider lines         */
   poke(GLCD_SEGMENT,RAMAddr+6+(Row >> 3),~(0x80 >> (Row & 0x0007)));
}
```

... Four...

The Hitachi LM215XB 480×128 panel is a leopard of a different spot: it uses half the data, runs at half the speed, and sprinkles dots all over RAM in four quadrants. Listing 8 shows the rather complex test pattern code. Remember to select the half-speed **`Dot Clock/2`** in JP12 for this panel.

Each of its 64 logical rows requires 240 half-speed **`Dot Clock/2`** pulses, so the 82C54 behaves as though this panel has 480 **`Dot Clock`** cycles per row. The panel uses the data from odd-numbered RAM addresses, putting 2×240×64 = 30720 pulses of the regular 480 ns **`Dot Clock`** in each frame, but we must duplicate the data in successive even- and odd-numbered bytes to keep the output of the **`LCD Data Latch`** stable during both regular **`Dot Clock`** cycles.

The test code iterates through all 128 visible rows. Displaying the row number requires the routines shown in Listing 9, because the layout separates the RAM data. The complexity involved in addressing the individual dots makes this code considerably harder to follow.

The `WriteBit` function computes `RAMAddr` based on the dot's row and column location. The `Align` variable then specifies the bit location within that byte, which

Listing 8
The Hitachi LM215XB uses a half-speed 960 ns Dot Clock and directs the four data bits to separate quadrants. This code duplicates the dots in successive even and odd RAM addresses to keep the data stable during the extended clock cycle.

```
outpw(GLCD_CTLS,GLCD_OFF);                          /* disable clocking      */
LoadTimer(1,0x04,480,I8254_BASE);                   /* Dot Clocks/row        */
LoadTimer(2,0x04,30720,I8254_BASE);                 /* total clocks/frame    */
outpw(GLCD_CTLS,GLCD_ON | GLCD_BLFAST);             /* enable & set blinking */
for (Row = 0; Row < 64; Row++) {
   for (Col = 0; Col < 240; Col++) {
      RAMAddr = 2 * (240 * Row) + Col;
      poke(GLCD_SEGMENT,RAMAddr+Col,0x00);          /* even address          */
      poke(GLCD_SEGMENT,RAMAddr+Col+1,0x00);        /* odd address           */
   }
}
for (Row = 0; Row < 128; Row++) {
   RAMAddr = 2 * (240 * Row) + 1;
   StoreByte(Row,0,BLINK_OFF,Row);
   StoreByte(Row,8,BLINK_FG,0x81);
   StoreByte(Row,16,BLINK_OFF,RAMAddr >> 8);
   StoreByte(Row,24,BLINK_OFF,RAMAddr);
   StoreBit(Row,32+Row,BLINK_FG,1);

   StoreByte(Row,240+0,BLINK_OFF,Row);
   StoreByte(Row,240+8,BLINK_BG,0x7E);
   StoreByte(Row,240+16,BLINK_OFF,RAMAddr >> 8);
   StoreByte(Row,240+24,BLINK_OFF,RAMAddr);
   StoreBit(Row,240+32+Row,BLINK_FG,1);
}
```

obviously depends on how you wire the Graphic LCD Interface's data bits to the LM215 connector. The `Blink` parameter controls the corresponding bit in the high nybble.

`WriteByte` simply calls `WriteBit` for each bit in a byte-sized value. This process entails considerable overhead, but remains reasonably perky because the panel has so few dots… and things will certainly go better with assembler!

The LM215 test pattern displays the row and `RAMAddr` values in all four panel quadrants to verify all the data bits. If you have one of these panels, examine the dots (with a magnifying glass?) to see that the same RAM address appears four times, once in each quadrant, on the proper lines.

... and More!

That covers the interesting LCD panels in my stash, without coming close to exhausting the possibilities. You can probably adapt the Graphic LCD Interface to drive whatever panel you've got, although some are sufficiently bizarre that they just won't work. As a rule of thumb, the more complex the panel, the older it's likely to

Listing 9

The Hitachi LM215XB panel has a peculiar bit arrangement that makes these two routines far more complex than you'd expect for such a small panel. The low nybble of each odd-numbered RAM address holds the normal dots, the high nybble holds the blinking dots, and each bit drives a separate quadrant. The StoreByte function distributes each bit to the proper location, making each incoming byte seem contiguous in storage.

```
StoreBit(Row,Col,Blink,Data)
WORD Row;
WORD Col;
WORD Blink;
WORD Data;
{
WORD Temp;
WORD Mask;
WORD RAMAddr;
WORD Align;

   RAMAddr = 2 * (240 * (Row % 64) + (Col % 240));/* even address          */
   Align = ((Col > 239) ? 2 : 0) + ((Row > 63) ? 1 : 0);
   Mask = 0x0001 << Align;
   Data &= 0x0001;

   Temp = peek(GLCD_SEGMENT,RAMAddr + 1);         /* fetch the old bits     */
   Temp &= ~(Mask | (Mask << 4));                 /* strip target & blink   */
   Temp |= Data << Align;                         /* insert new data bit    */

   switch (Blink) {
   default :
      putstr("Invalid blink mode in Storebit: %u\n",Blink);
   case BLINK_OFF:
      Temp |= Data << (Align + 4);                /* no blink dupe data bit*/
      break;
   case BLINK_FG :                                /* zero blink bit, done   */
      break;
   case BLINK_BG :
      Temp |= Mask << 4;                          /* set the blink bit      */
      break;
   case BLINK_INV :
      Temp |= ((~Data) & Mask) << 4;              /* inverse of data bit    */
      break;
   }
   poke(GLCD_SEGMENT,RAMAddr  ,Temp);
   poke(GLCD_SEGMENT,RAMAddr+1,Temp);
}


StoreByte(Row,Col,Blink,Data)
WORD Row;
WORD Col;
WORD Blink;
WORD Data;
{
   StoreBit(Row,Col  ,Blink,Data >> 7);
   StoreBit(Row,Col+1,Blink,Data >> 6);
   StoreBit(Row,Col+2,Blink,Data >> 5);
   StoreBit(Row,Col+3,Blink,Data >> 4);
   StoreBit(Row,Col+4,Blink,Data >> 3);
   StoreBit(Row,Col+5,Blink,Data >> 2);
   StoreBit(Row,Col+6,Blink,Data >> 1);
   StoreBit(Row,Col+7,Blink,Data     );

}
```

be: more recent panels have better LSI chips and simpler interfaces. Things just keep getting better and better!

Remember that you're limited to 32 KB of RAM, unless you piggyback another chip atop the **`LCD Refresh RAM`** or modify the circuit to use a 62512-type static RAM. If you find a panel requiring more memory than that, you'll probably wind up with a paged refresh buffer that fits within a single 64 KB memory segment. Writing the code for *that* should be quite a challenge…

Release Notes

The test code for this chapter, `GraphLCD`, exercises the Graphic LCD Interface hardware and generates test patterns for several common panels. You can use my code as a template to check out other panels with different specs.

Assuming that you can get the power supply and signal wires connected properly, `GraphLCD` can help you figure out the panel's sync requirements. The panels can tolerate incorrect signals (at least for a while), if you don't exceed their voltage specs, so try a few experiments. In fact, once you get a panel working, lie to `GraphLCD` just to see what the results look like.

CAUTION: even though the panels won't detonate if you apply the wrong sync signals, a DC bias will gradually degrade both the liquid crystal material and the panel's transparent electrodes. Turn the power off while you dope out the jumpers and analyze the clocking. Don't use incorrect signals longer than it takes you to realize that things just aren't working right.

So… if that doesn't get your juices flowing, you really are reading the wrong book. Heat up those soldering irons, get those compilers whirring, and let's see some dots!