16 All Text Is Graphics

Have you noticed how much the few remaining DOS text-mode programs resemble Windows apps? Otherwise sensible companies spend precious programmer hours cross-dressing their programs… as though 2000 characters allowed enough screen real estate for fancy borders and pushbuttons.

Ah, well, such is progress.

This chapter marks, perhaps, a step in the same backwards direction. We're now going to turn a perfectly serviceable, bitmapped, graphics LCD panel into a moderately intelligent Glass Teletype with ANSI cursor positioning and attribute controls. We can then display status information without using a serial port or the video screen. If your project requires status output that *can't* appear on a standard VGA-compatible screen (perhaps because your embedded project puts the user interface on the VGA), this may be just what you need.

First, we'll see where the character bitmaps come from, then cover the assembler code that shuffles them into the **`LCD Refresh RAM`**. Next come the TTY routines that decode standard ASCII control characters and ANSI commands to provide full screen positioning.

I'll wrap up with a chilling murder mystery, The Case of the Capital T, that shows why you can't take anything for granted. Hint: you already know both the problem and the solution…

Filching Fossil Fonts

The Original IBM PC had two display options: the text-only Monochrome Display Adapter and the bitmapped Color Graphics Adapter. You could install both boards in the same system: the MDA presented high quality text on one screen and the CGA displayed moderately good graphics on another.

The catch was that, in graphics mode, the CGA hardware didn't support text output. The CPU painstakingly drew every character, dot by dot, whenever a program wrote text on the screen. Even though those functions *really* belonged with the CGA hardware, the PC's designers included them in the BIOS ROMs on the system board. The architecture for installable BIOS extensions came somewhat later, after the IBM PC/XT hit the scene.

Now, nearly two decades later, the PC Compatibility Barnacles dictate that every PC BIOS must include those ancient CGA functions, even though the standard

video board has become a Super VGA accelerator with its own BIOS extension. You may have the latest SuperTurbo 100 MegaWinMark video accelerator, but your PC's system board BIOS stands ready for the day when you reinstall that good old CGA. It could happen...

The CGA presented 640×200 dots in one color on a background of another color, which means anything it could do maps directly to our 640×200 LCD panels, at least if we're willing to give up those colors. In particular, the BIOS includes an 8×8 dot character font that, while it may not have the nicest looking characters in the world, lies waiting for our firmware to put it to good use.

We cannot, unfortunately, use the BIOS character drawing routines. The CGA used a simple, linear memory map quite different from our LCD panels, which, as you've seen, distribute dots from a single byte into widely separated locations on the panel. Worse, each panel has a different memory map that precludes one-size-fits-all firmware.

The Original IBM PC put those font bitmaps at address F000:FA6E. The PC Compatibility Barnacles guarantee that address remains valid in every PC clone ever built. The bitmap for a particular character starts at:

```
F000:FA6E + 8×(character value)
```

and occupies eight bytes. You can see how the process works in Listing 1, which copies a single character from the bitmap into the **`LCD Refresh RAM`**.

Listing 1
This C function writes a character bitmap, a.k.a. the character's glyph, into the LCD Refresh RAM. Because the assembly language routines hide all the panel-specific bit twiddling, this routine remains the same for all the LCD panels. The font width in our code is always eight bits, the same as the BIOS bitmaps, just to keep things simple.

```
void LCDWriteGlyph(int CharCode,int DotRow,int DotCol) {
int ScanLine;

   if ((CharCode < 0) || (CharCode >= NumFontChars)) {
      CharCode = '?';                              /* an obvious filler char*/
   }

   CharCode *= FontHeight;                         /* point to start of char*/

   for (ScanLine=0; ScanLine < FontHeight; ++ScanLine) {
      LCDWriteByte(*(pFont+CharCode+ScanLine),
          DotRow+ScanLine,DotCol,BlinkMode);
   }
}
```

The BIOS bitmaps include only the first 128 characters (0x00 through 0x7F), so `LCDWriteGlyph` substitutes a question mark if the character value exceeds 127. It then iterates through the eight bitmap rows, copying one byte at a time. In this case, `FontHeight` and `FontWidth` must both equal eight dots, as the BIOS table knows no other character size.

If you want different, smaller, or very detailed characters, you can modify this routine to copy the appropriate bits from your own custom bitmap table to the **`LCD Refresh RAM`**. This also requires changes in the `LCDWriteByte` routine that handle partial bytes, should your glyphs not be a multiple of eight bits wide.

On the other hand, double size characters, at least *ugly* double size characters, are comparatively easy. Just modify the code to duplicate each dot from the font table into two adjacent bits and duplicate that byte in the next LCD row, wherever that row may fall in the buffer.

The BIOS ROMs may not be the only source of bitmap characters in a stock ISA bus PC: any graphics board compatible with the VGA includes a BIOS extension with the fonts appropriate for it, ready for use. IBM's EGA and VGA adapters standardized a method for locating these bitmaps, giving you a wider choice of characters if your embedded PC has such a board in it. Because our LCD code operates entirely independently of the built-in BIOS, you can use any of the bitmaps without regard to the current BIOS video mode or display resolution.

The corresponding Bad News is that there's no tidy way to decide if the target system contains an MDA, a CGA, a VGA, an SVGA, or no video board at all. The code shown in Listing 2 checks the `INT 1Fh` vector, which *should* point to the bitmap for character 0x80 if the font can support more than 128 characters, which is the case for all video adapters other than the CGA.

The vector should not point to an actual interrupt handler, although some BIOSes seem to aim it at an unexpected-interrupt handler during their power-on self tests and leave it set if they don't find a video board. As a result, your code cannot assume that a nonzero value indicates a valid pointer to a font table. I know which board I have installed here and I'll leave you to experiment on your own target system.

The algorithms I've seen in other books assume that the system will have at least one video adapter, so they don't test for the possibility of no adapter at all. If you adapt a standard algorithm, make sure it doesn't fall into the same trap.

If the code finds a nonzero `INT 1Fh` vector, it issues `INT 10h, AX=1130h` to find the address of one of the board's font bitmaps. VGA boards include both 8×8 and 8×16 fonts, the latter of which works eminently well with 400-line LCD panels.

Listing 2
This routine sets up the font bitmap for the rest of the code. If the system has no video board, the INT 1Fh vector should be 0000:0000 to indicate that the default 128 character CGA font is in effect. That vector will be nonzero if an EGA or VGA board is installed, in which case Int 10h, AX=1130h returns a pointer to the desired font table. I used the 8x8 font for 200 line panels and the 8x16 font for larger panels to show that more dots can produce a slightly less ugly font. If you prefer more information over prettier characters, use the 8x8 font regardless of the panel size.

```
void LCDInitFont(void) {
struct REGPACK rp;
void far *Int1FVector;
int VidMode;

/*--- figure out what character table is available                          */

   puts("Initializing character generator...");

   Int1FVector = (void far *)*(long far *)MK_FP(0x0000,4*0x1F);
   printf("  Int 1F vector is %Fp\n",Int1FVector);
   if ((void far *)NULL == Int1FVector) {         /* have chars 80..FF?    */
      puts("  Using ROM BIOS CGA font");
      pFont = (BYTE far *)MK_FP(0xF000,0xFA6E);   /*  nope, use BIOS ROM   */
      FontHeight = 8;                            /*  which is 8x8 only    */
      FontWidth = 8;
      NumFontChars = 128;                        /*  basic ASCII only!    */
   }
   else {
      puts("  Using video card font");
      VidMode = *(BYTE far *)MK_FP(0x0040,0x0049);
      printf("  ... resetting video mode: %u\n",VidMode);

      rp.r_ax = (0x00 << 8) | VidMode;           /* Reset video mode      */
      intr(0x10,&rp);

      rp.r_ax = 0x1130;                          /* Get Font Pointer info */
      if (NumDotRows > 200) {
         rp.r_bx = 0x0600;                       /*  BH = 06 for 8x16 font*/
         FontHeight = 16;
      }
      else {
         rp.r_bx = 0x0300;                       /*  BH = 03 for 8x8 font */
         FontHeight = 8;
      }
      printf("  ... fetching %u line font pointer\n",FontHeight);
      intr(0x10,&rp);

#if 0
      printf("ax %04X bx %04X cx %04X dx %04X ds %04X es %04X bp %04X\n",
         rp.r_ax,rp.r_bx,rp.r_cx,rp.r_dx,rp.r_ds,rp.r_es,rp.r_bp);
#endif

      pFont = (BYTE far *)MK_FP(rp.r_es,rp.r_bp);
      FontWidth = 8;
      NumFontChars = 256;
   }

   printf("  pFont=%Fp, FontHeight=%u, NumFontChars=%u\n",
          pFont,FontHeight,NumFontChars);
```

Listing continues on next page

Listing continues from previous page

```
/*--- set up screen control variables                                    */

   NumCharRows = NumDotRows / FontHeight;
   NumCharCols = NumDotCols / FontWidth;
   VScrollAllowed = 1;

   printf("  Panel has %u character rows and %u character cols\n",
          NumCharRows,NumCharCols);

   return;
}
```

You'll quickly see that with more dots available the characters don't look *quite* so ugly. The code then sets several variables detailing the font's height, width (always eight bits for now), and the number of characters in the bitmap.

This should work with VGA video boards, but I won't be surprised to find that it fails in systems with other display boards. In particular, EGA boards don't include the 8×16 font table. I much misdoubt any of you still have an EGA; if you do, you must tweak the code to support 400-line panels. In any event, should strange things happen, enable the `#if 0` code stub in Listing 2 to dump the registers returned by `INT 10h` and see what went wrong.

Incidentally, I used the `intr()` function from the Borland C library, rather than the more familiar `int86x()`, because the BIOS function returns the font pointer in `ES:BP`. Check your compiler doc, but as near as I can tell, `intr()` provides the only way to examine `BP` without resorting to inline assembler.

You can short circuit the entire font search routine with your own code to install a completely custom font. Keep in mind that the rest of the code assumes that the bitmaps are eight bits wide, with the sky the limit in the other direction.

Also remember that the **`LCD Refresh RAM`** contains bitmap patterns, not the ASCII character values. Although the CGA included a screen readback function that searched the font bitmap table to find the pattern matching the one at the cursor location, I didn't attempt to duplicate that stunt. If you *must* read characters back from the **`LCD Refresh RAM`**, I suggest allocating a 2000 (or 4000, or whatever) byte buffer and storing the ASCII characters in parallel with the bitmaps. Trust me, it's a *whole* lot easier and faster than pattern recognition.

Now that we have a source of character bitmaps, the next step drops them into the **`LCD Refresh RAM`**. As you will recall from Chapter 15, most LCD panels are not nearly so accommodating as the good old CGA.

Divvying up the Dots

Listing 3 shows `LCDWriteByte`, a function that writes a single byte into the DMF651 **`LCD Refresh RAM`**. The character coordinates use dot increments, but the column must be a multiple of eight to position the characters correctly. Similarly, positioning the character rows on multiples of the font height produces a regular vertical spacing.

Listing 3
The fundamental character operation involves writing a single byte into the LCD Refresh RAM at a specific row and column. This DMF651 routine computes the buffer address, splits the byte into two nybbles, and sets the blinking bits for each. Aligning characters on 8-dot column boundaries considerably simplified the code!

```
                PUBLIC    C LCDWriteByte
                PROC      LCDWriteByte
                ARG       CharCode:WORD,DotRow:WORD,DotCol:WORD,Blinking:WORD
                USES      ES,DI

                MOV       AX,[DotCol]         ; set up column number
                AND       AX,NOT 0007h        ;  ... force byte alignment
                MOV       BX,[DotRow]         ; set up row number
                CALL      LCDMakeCharAddr

                LES       AX,[pLCDBuff]       ; get buffer base pointer
                ADD       DI,AX               ;  ... update byte pointer

;--- insert the left nybble and blinking pattern

                MOV       AX,[CharCode]       ; fetch new byte value
                MOV       DL,AL               ; save high bits for later
                SHR       AL,4                ; align nybble, clear blink bits

                CMP       [Blinking],1
                JE        SHORT @@1
                MOV       AH,DL               ; no blinking, copy data bits
                AND       AH,0F0h
                OR        AL,AH
@@1:
                MOV       [ES:DI],AL          ; wr blinking + data to buffer

;--- insert the right nybble and blinking pattern

                INC       DI                  ; step to next buffer byte
                AND       DL,00Fh             ; get nybble, clear blink bits

                CMP       [Blinking],1
                JE        SHORT @@2
                MOV       BH,DL               ; no blinking, copy data bits
                SHL       BH,4
                OR        DL,BH
@@2:
                MOV       [ES:DI],DL          ; write blinking + data to buffer

                RET

                ENDP      LCDWriteByte
```

The Graphic LCD Interface supports blinking characters on the DMF651 panel. The code splits each byte from the BIOS bitmap in half and stores the fragments in the low nybbles of two successive buffer bytes. Because the hardware accepts four bits at a time, we can update the bytes without using the direct dot plotting code from Chapter 15.

The **LCD Data Multiplexer** displays bits 4:7 alternately with bits 0:3 from each byte, switching between them at the selected blink rate. To suppress visible blinking, the high and low nybbles in each byte must be identical. If one nybble holds zero, the dots in the other nybble appear to blink on and off against a blank background. You can implement other blinking schemes if you like, as well as inverse characters and so forth.

If you want unaligned or proportionally spaced characters on your LCD panel, your code must handle bit patterns that don't fit neatly into each byte. In fact, a single bitmap byte may affect three successive **LCD Refresh RAM** bytes. The code gets particularly tricky near the right edge of the panel, where a character may run off the end of the panel. Should you wrap, clip, or ignore that character?

Aligned characters also simplify the buffer address calculation routine. Listing 4 shows the few lines of `LCDMakeCharAddr` for a DMF651 panel. Recall that the dot-drawing `LCDMakeAddr` routine in Chapter 15 set `CL` to the value that shifted a dot to or from the low-order bit. Now, because we always fill the entire nybble with data, we need no further shifting.

Adapting this code to other panels should be reasonably straightforward. For example, the 640×400 LG64AA44D panel accepts eight bits on each **Dot Clock**

Listing 4
This routine for a DMF651 panel computes the LCD Refresh RAM address corresponding to a character's location given as a dot column in AX and a dot row in BX. The characters always align on an 8x8 dot grid, eliminating the shift amount computation.

```
PROC      LCDMakeCharAddr

ADD       BX,BX               ;  make word table index
MOV       DI,[RowStarts+BX]

DIV       [BYTE ColModulus]   ; col / modulus
MOV       CL,0                ; force shift amount to zero
MOV       AH,0
ADD       DI,AX               ; DI points to the byte

RET

ENDP      LCDMakeCharAddr
```

and, thus, doesn't support blinking. Bits 0:3 of each byte appear on rows 0 through 199, while bits 4:7 fill rows 200 through 399. You must insert the new data into each nybble without disturbing the existing bits in the *other* nybble of the byte.

Listing 5 shows the LG64AA44D `LCDWriteByte` function. For this panel, `LCDMakeCharAddr` sets `CL` to align the new nybble in either the low or high half of the **`LCD Refresh RAM`** byte, depending on which row it occupies. The code reads the existing byte from the buffer, strips out the old bits, inserts the new ones,

Listing 5
The LG64AA44D LCD Refresh RAM layout is more complex than the DMF651. Because the 400-row panel accepts and displays eight bits on each Dot Clock, the hardware cannot support blinking. This routine splits a byte into two nybbles and inserts them into the appropriate parts of the two target bytes. As with the DMF651, the character's column address must be a multiple of 8.

```
                PUBLIC    C LCDWriteByte
                PROC      LCDWriteByte
                ARG       CharCode:WORD,DotRow:WORD,DotCol:WORD,Blinking:WORD
                USES      ES,DI

                MOV       AX,[DotCol]           ; set up column number
                AND       AX,NOT 0007h          ;  ... force byte alignment
                MOV       BX,[DotRow]           ; set up row number
                CALL      LCDMakeCharAddr

                LES       AX,[pLCDBuff]         ; get buffer base pointer
                ADD       DI,AX                 ;  ... update byte pointer

;--- insert the left nybble

                MOV       AX,[CharCode]         ; fetch new byte value
                MOV       DL,AL                 ; save high bits for later
                SHR       AL,4                  ; set up for alignment
                SHL       AL,CL                 ; align to target nybble
                MOV       BH,0F0h               ; set up bit mask
                ROL       BH,CL                 ;  ... align to strip old bits

                MOV       AH,[ES:DI]            ; fetch target bits
                AND       AH,BH                 ; strip old bits
                OR        AL,AH                 ; tuck in new bits
                MOV       [ES:DI],AL            ; put 'em back in the buffer

;--- insert the right nybble
;    this uses the same nybble mask because it's in the same screen half

                INC       DI                    ; step to next buffer byte
                AND       DL,00Fh               ; extract low nybble
                SHL       DL,CL                 ; align to target nybble

                AND       BH,[ES:DI]            ; fetch target bits, strip old
                OR        DL,BH                 ; tuck in new bits
                MOV       [ES:DI],DL            ; put 'em back in the buffer

                RET
                ENDP      LCDWriteByte
```

and writes the modified byte back into the same buffer address. Although it's slightly slower than the DMF651 code, you won't notice the difference in real life.

So much for a single character. Now, to put them where we want…

Gazing in the Glass

First of all, a confession. I have not implemented a real blinking character cursor for the LCD panel, because I use it as an output-only status device. Nevertheless, we must have a name for the spot where the next character will appear and *current cursor location* seems as good as any. If you want a visible cursor at the current cursor location, well, it's a simple matter of firmware: load a blinking block and be done with it! Of course, if your hardware doesn't support blinking, you may have a bit more difficulty.

Extra credit project: implement blinking on an 8-bit panel without using the dot multiplexer. Hint 1: you can hitch an interrupt routine to the **`Frame Sync`** pulse, count off refresh intervals, and rewrite the character at the cursor position on the fly. Reread the discussion of output-only ports in Chapter 9 before you proceed, to avoid winding up with two routines clobbering each other's output bit patterns. Hint 2: consider the dots on the top row before you write the interrupt handler.

The ASCII character set includes about 32 control characters that are not ordinarily displayed. Instead, they affect how the remaining 96 characters appear on paper, the current rage when that character set coalesced into a standard. The control characters include such old friends as **Carriage Return** and **Line Feed**, as well as obscure relatives like **End of Medium**.

Because the font tables in the BIOS and VGA ROMs include visible bit patterns for all those characters, your code may ignore their control functions and simply display them as ordinary, albeit funny looking, text.

Which control character functions you implement and exactly what they do depends on how thorough you are. Nobody (well, hardly anybody) uses paper output any more, making the mappings into video functions somewhat arbitrary. The term *Glass Teletype* pretty well describes the level of control you can achieve: no matter how hard you try, the results still look a lot like a really short roll of paper.

Listing 6 shows the bare bones implementation I chose for the LCD panel code. **Line Feed** and **Carriage Return** do pretty much what you'd expect, **Form Feed** clears the panel, **Back Space** moves the cursor one space to the left, **Tab** moves it rightward to the next multiple of four columns, and **Delete** erases the character at the cursor

Listing 6
Emulating a Glass Teletype requires picking control characters out of the incoming data that can affect either the current cursor location or the LCD Refresh RAM contents. This code handles a basic set of controls that suffice for simple text output applications. The VScrollAllowed variable defaults to TRUE, so that a Linefeed on the bottom row of the panel scrolls the entire panel contents up by one character row.

```
void LCDSendChar(int CharCode) {

   switch (CharCode) {
   case LF :
      if (CurDotRow >= (NumDotRows - FontHeight)) {/* last line?           */
         if (VScrollAllowed) {
          LCDScrollUp(FontHeight);                /* yes, scroll up a line */
          CurDotRow = NumDotRows - FontHeight;    /* force to last line    */
         }
         else {
          CurDotRow = 0;                          /* no scroll, wrap to top*/
         }
      }
      else {
         CurDotRow += FontHeight;                 /* no, so step down      */
      }
      break;

   case CR :
      CurDotCol = 0;                              /* to leftmost col       */
      break;

   case FORMFEED :
      LCDClear();
      break;

   case BS :
      if (CurDotCol >= FontWidth) {               /* if not at edge        */
         CurDotCol -= FontWidth;                  /*  back up one column   */
      }
      break;

   case TAB :                                     /* tab to next stop      */
      do {
         LCDSendChar(' ');
      } while (((CurDotCol/FontWidth)+1) % TABINTERVAL);
      break;

   case DEL :
      LCDScrollLeft(CurDotRow,CurDotCol,FontHeight,FontWidth);
      break;

   default :
      LCDWriteGlyph(CharCode,CurDotRow,CurDotCol);    /* display it        */
      CurDotCol += FontWidth;                     /* tick column counter   */
      if (CurDotCol > (NumDotCols - FontWidth)) { /* past end of line?*/
         CurDotCol = 0;                           /* reset to left edge    */
         if (!HWrapAllowed) {
          LCDSendChar(LF);                        /* and step to next line */
         }
      }
   }

}
```

position. The code treats everything else as a displayable character and passes it to the LCD panel character routine.

The `CurDotRow` and `CurDotCol` variables hold the current cursor location in terms of the dot coordinate of the character's upper left corner. Although I force the characters to lie in a grid the size of the font bitmap, you can tweak the code to handle multiple character sizes and other effects.

The code supports vertical scrolling, displaying the last 25 (or 50, or whatever) lines sent to the panel. I won't show the code here, but, contrary to what you might think, you can't just schlep the bits around with a single `REP MOVSB` instruction. As an example, consider moving characters on an LG64AA44D panel, where the same byte can hold both source and target dots. Not a pretty sight, I assure you.

The **Delete** character requires a routine that removes a rectangular block of dots by shifting the remaining characters in the line leftward by one position. In that case, you must copy a group of dots to a single position, then repeat that for each row in the character while filling the right end of the line with blanks. I did not include a corresponding **Insert** function, but you can add it fairly readily should your application require it.

In today's embedded systems, most of the output characters will come from C strings, so I included an `LCDSendString` function that handles the familiar zero-terminated strings. You can embed control characters in the strings or send them directly to `LCDSendChar` as needed. Perforce, ASCII character 0 (that's a binary zero, not 0x30) becomes a nondisplayable control character.

The code also includes several utility functions that enable and disable vertical scrolling, end-of-line wrapping, and so forth. Examine the source code on the diskette, if you're interested in this sort of thing (as you *should* be, having gotten this far in the book!)

One thing you *can't* do with ASCII control codes, though, is change the position of the next output character without affecting the existing text. For example, once you reach the last row of the screen, you can't get back to the top without sending a **Form Feed** character that clears everything. Seems a bit extreme, doesn't it?

Presenting a nicely formatted status display using only ASCII control codes is nearly impossible, although you've surely seen some noteworth attempts at pulling it off. Fortunately, the ANSI standards committee set down some rules for cursor control on serial display terminals that we can adapt to our graphics LCD panels. The end result is well worth the programming effort.

Assume the Position...

You're probably familiar with ANSI control sequences, if only from their use in PC-DOS **PROMPT** strings. For our present purposes, I'll simply say that they allow the same output string to produce similar results on anything that can interpret the sequences. Because they're a genuine standard (pretty much), you'll find lots of *anythings* that can interpret them.

Figure 1 summarizes the control sequences used in the Graphic LCD character interface. Each sequence starts with the same two characters: 0x1B (**Escape**) and 0x5B (the left square bracket: **[**). Next come any numeric parameters, represented with ASCII digits, and the terminating letter that identifies the command. The case of that final letter is significant: **ESC[2j** puts the cursor at row 2, column 1 while **ESC[2J** clears the panel and homes the cursor. Pay attention to your typing, OK?

Figure 1
All ANSI Cursor Control Sequences begin with two common characters:

ESC ASCII 27, the Escape character
[ASCII 91, the left square bracket

There are no blank characters within these command strings. Numeric parameters, represented by an octothorpe (#), use ASCII decimal notation (characters 0x30 through 0x39) and default to 1 if omitted. Row and column numbers start with 1 at the upper left. The upper/lower case of the trailing letter is significant!

```
Command       Example   Function

ESC[#A        ESC[2A    Cursor up # rows (up 2)
ESC[#B        ESC[B     Cursor down # rows (down 1)
ESC[#C        ESC[10C   Cursor right # columns (right 10)
ESC[#D        ESC[5D    Cursor left # columns (left 5)

ESC[#;#H      ESC[H     Set cursor to row;column (default 1,1)
ESC[#;#f      ESC[1;2f  Set cursor to row;column (1,2)
ESC[#;#j      ESC[3j    Set cursor to row;column (3,1)
                        (H,f, and j are all synonyms)

ESC[s         ESC[s     Save current cursor location (1 level)
ESC[u         ESC[u     Restore saved cursor location

ESC[2J        ESC[2J    Clear display and home cursor

ESC[K         ESC[K     Clear from cursor to end of row

ESC[#h                  Set display mode, ignored except for:
              ESC[7h     wrap at end of each row

ESC[#l                  Reset display mode, ignored except for:
              ESC[7l     no wrap at end of row

ESC[#m                  Set display attributes, ignored except for:
              ESC[0m     disable all attributes
              ESC[7m     set blinking attribute
```

Decoding these sequences can be somewhat difficult, because the numeric parameters for some commands may be omitted or replicated. The state machine shown in Listing 7 tracks the possible combinations and calls a decoder routine when it encounters the final command character. Every character sent to the display *must* pass through this routine, imposing a moderate performance penalty for the privilege of standardized cursor control.

The actual ANSI function routines are nearly anticlimactic, as you can see in Listing 8. Most require just a line or two of code that adjusts the cursor position or sets a control variable to affects subsequent characters. Adding new commands is also easy: just insert a new `switch case` clause and the requisite code.

One conflict between ANSI cursor controls and Glass Teletype mode occurs when you write a character into the extreme lower right corner: row 25, column 80 on a 200-line display. In TTY mode, the panel scrolls upward by the font height to make room for the next line of characters, discarding the top row of characters in the process. Because ANSI controls generally create a static display image, perhaps

Listing 7
Decoding the ANSI control sequences depends on a state machine to track all the allowable parameters and command terminators. Each new character sent to the LCD panel invokes this function. It extracts the numeric parameters and passes control to the function decoder when it encounters the terminating letter in the command.

```
void LCDAnsiDriver(int NewChar) {

   ANSIBuffer[ANSICharCtr] = NewChar & 0x7F;      /* strip & save          */
   ++ANSICharCtr;
   ANSICharCtr = min(ANSICharCtr,ANSI_MAXLEN-1);

   switch (ANSIState) {
   case ANSI_WAIT :                    /* waiting for ESC char             */
      ANSICharCtr = ANSIParamCtr = 0;             /* force buffer restart  */
      if (ESC == NewChar) {
         memset(ANSIBuffer,0,sizeof(ANSIBuffer));
         ANSIState =  ANSI_MODE;
      }
      else {
         LCDSendChar(NewChar);                    /* just dump it          */
      }
      break;
   case ANSI_MODE :                                /* expect [ as next char */
      if ('[' == NewChar) {
         ANSIState =  ANSI_PARAM;
      }
      else {
         LCDAnsiNull(NewChar);                    /* ditch sequence        */
      }
      break;
```

Listing continues on next page

Listing continued from previous page

```
   case ANSI_PARAM :
      if ('=' == NewChar) {
         break;                                  /* gobble this one...    */
      }
      else {
         if (isdigit(NewChar)) {
           ANSIState =  ANSI_NUM;                /* still a number        */
         }
         else {
          if (isalpha(NewChar)) {
            LCDAnsiGetNumber();                  /* get number if any     */
            LCDAnsiDoCmd(NewChar);               /* end of sequence       */
          }
          else {
            LCDAnsiNull(NewChar);
          }
         }
      }
      break;
   case ANSI_NUM :
      if (';' == NewChar) {                      /* end of param digit    */
         LCDAnsiGetNumber();                     /* save prev param       */
      }
      else {
         if (isdigit(NewChar)) {
           ANSIState =  ANSI_NUM;                /* still a number        */
         }
         else {
           if (isalpha(NewChar)) {
              LCDAnsiGetNumber();                /* get number if any     */
              LCDAnsiDoCmd(NewChar);             /* end of sequence       */
           }
           else {
              LCDAnsiNull(NewChar);
           }
         }
      }
      break;
   default :
      LCDAnsiInit();                             /* error, restart decoder*/
   }
}
```

with a tidy Windows-ish border around the whole screen, you definitely *don't* want vertical scrolling when you write a character in that position!

You can enable line wrapping by sending **ESC[7h** and disable it with **ESC[7l** around the offending characters, but that's a nuisance. Because I plan to use the panel as a static display, I simply disable vertical scrolling in the ANSI initialization routine. I can't find an ANSI control sequence to handle this, but I won't be surprised if one of you folks comes up with the Official Doc defining that command.

So, there you have it: a reasonably fast, large, and cheap character output device that leaves all the standard PC hardware untouched. Now you can run your

embedded PC program, use the target system's serial port or video display as usual, and still present rapid, intelligible diagnostic information for your own use without affecting anything else.

Listing 8

This excerpt from the ANSI function decoder shows that handling the commands is mostly a matter of a few calls to routines in the LCDChars module that twiddle the cursor location. We absorb the unused ANSI functions to prevent confusing the LCD panel's code with functions it cannot perform.

```
void LCDAnsiDoCmd(int NewChar) {

BYTE Counter;

   ANSIState = ANSI_WAIT;              /* by default...                    */

   switch (NewChar) {

   case 'A' :                          /* cursor up                        */
      LCDSetCharCursor(LCDGetCharRow()-ANSIParams[0],LCDGetCharCol());
      break;
<<< cases omitted >>>
   case 'H' :                          /* set cursor position*/
   case 'f' :                          /*   (synonym)                      */
   case 'F' :                          /*   (synonym)                      */
      for (;ANSIParamCtr < 2; ++ANSIParamCtr) {
         ANSIParams[ANSIParamCtr] = 1;           /* force defaults         */
      }
      LCDSetCharCursor(ANSIParams[0]-1,ANSIParams[1]-1);/* fix origin!     */
      break;

   case 'J' :                          /* erase, home cursor               */
      if (2 == ANSIParams[0]) {
         LCDClear();
      }
      break;

   case 'K' :                          /* erase to end of line             */
      Counter = NumCharCols - LCDGetCharCol();
      while (Counter) {
         LCDSendChar(DEL);
         --Counter;
      }
      break;
<<< cases omitted >>>
   case 's' :                          /* save cursor location             */
      ANSISavedRow = LCDGetCharRow()+1;
      ANSISavedCol = LCDGetCharCol()+1;
      break;

   case 'u' :                          /* restore cursor location          */
      LCDSetCharCursor(ANSISavedRow-1,ANSISavedCol-1);
      break;

   default :
      LCDAnsiNull(NewChar);            /* dump all others...               */

   }
}
```

The Rest of the Stories

The 640×200 TLY-365-121 panel resembles the DMF651, although you must remember that each row contains 1280 dots. Tweak the DMF651 character address code supplied on the diskette and it'll work fine for you.

The 480×128 LM215, as always, presents a challenge. Because the four dots in each byte appear in different quadrants you must set one bit at a time. Drawing the dots for a single 8×8 character thus requires 64 calls to the `LCDSetDot` function we used in Chapter 15. Modify that code to support blinking if you really must have it. All this, just for a lousy 960 characters? I didn't think so, either, which is why you won't find any code for it on the diskette.

The 640×400 LM64015T should be a piece of cake because each bitmap byte appears in a single LCD Refresh RAM byte. I didn't write code for it, because, lacking both the backlight inverter and the desire to tinker with kilovolt power supplies, I can't see the results.

Now, to tie everything you've read in this book together, here's a terrifying tale...

The Case of the Capital "T"

After I'd finished the Graphic LCD Interface hardware described in this book, I used it to present debugging and tracing output from the *x*86 protected mode kernel I described in my *Firmware Furnace* column in *Circuit Cellar INK* magazine Issues 48 through 65. The LCD provided a convenient way to present rapidly changing status information without using the PC's video display.

However, when I wrote the protected-mode driver for the TLY365 panel and ran a set of test patterns, my target system developed a curious problem. Although the LCD panel worked fine, the system hung midway through the *next* BIOS boot sequence after I pressed the Reset button. Cycling the PC's power cured the problem, but it still hung reliably after every manual reset.

Hmmm...

Some scope probing revealed that the CPU was running in a tight loop doing a little I/O and a lot of memory writing. It surely wasn't any of my code because I didn't have any BIOS extensions installed. Just to make sure, I pulled the battery backed RAM out of the Firmware Development Board's socket. The CPU still got wedged after each reset.

For lack of a better idea, I pulled the Graphic LCD Interface's refresh RAM. As you might expect, the system worked perfectly. Swapping RAM chips didn't solve the problem, so it wasn't a faulty chip.

The system worked correctly with the DMF651 and LG64AA44D panels and the appropriate drivers drawing the same test pattern. The TLY365 worked OK with the Game of Life and the ANSI test code presented in this book. It failed *only* on a manual, pushbutton boot *after* displaying a text test pattern in protected mode.

I modified the test pattern and discovered that, with a completely blank LCD panel, the system booted normally. A quick divide-and-conquer search showed that the failure occurred when a capital **T** appeared in the first character position of line 12. No other characters on that line seemed to matter: the string **Tab stops...** failed in the same way as single **T** followed by blank characters.

I whipped out my jeweler's loupe and examined the panel, then drew up Figure 2 showing the bit patterns, dot row numbers, and RAM addresses for line 12. If you've been paying attention throughout this book, you should be able to spot the problem immediately.

This is a quiz!

Give up?

Twelve lines (0 through 11) of 8×8 BIOS character cells above line 12 means that the top bar of the **T** character lies on dot row 96 (numbered from 0 through 96, remember). Each dot row occupies 320 bytes of RAM, because the TLY365 has 1280 dots on each of 100 logical rows, arranged in a 640×200 physical array. Row 1 starts at address 0000, which puts dot row 96 at address (96-1) × 320 = 76C0.

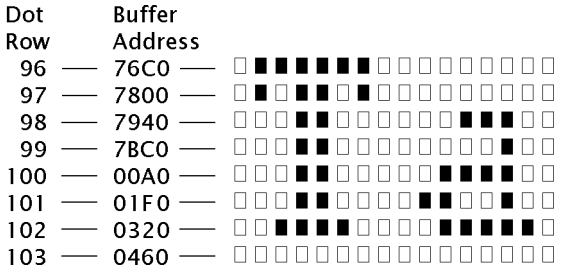

Figure 2
My protected-mode LCD test code wrote "Tab stops..." on line 12. This figure shows the LCD Refresh RAM addresses and bit patterns corresponding to the first two letters on a TLY365 640x200 panel. The Graphic LCD Interface hardware blinks by alternating the upper and lower nybbles of each byte, so every four dots on the panel require an 8-bit byte in the RAM.

Row 97, the *second* row of the character cell, begins at address 7800. The dot pattern for that row begins with the tips of the T's serifs and its two-dot central stroke. The hex value for that pattern is 5A, which should look suspicious already. Now, recall that the TLY365 accepts four bits of data on each **`Dot Clock`** and the Graphic LCD Interface implements blinking by alternating between the nybbles of each **`LCD Refresh RAM`** byte.

Because the **`T`** *isn't* a blinking character, the refresh buffer bytes at addresses 7800 and 7801 must have identical nybbles. Therefore, those two bytes are 55 AA.

Now, if *that* doesn't raise your hackles, you flunk...

The BIOS boot routine scans memory between C0000 and EFFFF for BIOS extensions after each reset. By definition, a BIOS extension starts on a 2 KB memory boundary with two flag bytes. Do you remember, from Chapter 6, that the BIOS extension flag bytes are 55 and AA?

The second character on line 12 was either a lowercase **`a`** or a blank, neither of which has any dots on row 97. That means a pair of binary zero bytes following the flag bytes. The *third* byte of a BIOS extension header gives the extension's length in multiples of 512 bytes.

So, the first three bytes of that line just happened to define a BIOS extension at address 7800 with a length of either zero or 512×256 = 128 KB. The fact that I didn't deliberately write an extension doesn't matter; the BIOS found those bytes during its memory scan and didn't inquire as to my intent.

Gotcha!

A valid BIOS extension includes a checksum byte that makes the sum of all the bytes in the extension equal to zero. Evidently, the BIOS checksum routine in my PC concluded that an extension of zero length, lacking any content, is always valid. My guess is that the BIOS recognized the header, but the checksum routine simply gave up when it saw a zero length byte.

So the situation goes a little something like this...

Immediately after turning the power on, the BIOS finds nothing particular in the **`LCD Refresh RAM`** and boots normally. My protected-mode code wrote its test pattern on the LCD and, quite inadvertently, plunked what looked like a BIOS extension header precisely at a 2 KB boundary within the refresh buffer.

When I pressed the system's Reset button, the BIOS ran through its extension scan again, whereupon it found the bogus header at address D000:7800. It erroneously concluded that the extension's checksum was valid and branched to offset 7803 in the **LCD Refresh RAM**. What happened after that was up for grabs, although we know the first byte of the "program" was a binary zero.

The solution was easy enough: disable the **LCD Refresh RAM** when the ISA bus **ResDrv** line goes active and keep it disabled until our firmware gains control. With the RAM disabled, the BIOS scan cannot find an extension in the buffer. Any program that uses the Firmware Development Board must enable the RAM before writing to the LCD.

You'll recall that we added a MAX691 watchdog timer with a latch to stretch its timeout interval after a reset. That guardian hardware has a signal that will keep the BIOS under control. Page back to Schematic 2 in Chapter 7 for the details.

The signal **+No Access Yet** from U18 goes high whenever **ResDrv** is active and returns low when the firmware writes to port 031C. That port contains the watchdog, character LCD, serial number, and other miscellaneous bits.

You might solve this another way, too. Because the BIOS scans in ascending address order, you could install a deliberate BIOS extension in the battery backed RAM. That extension, which would get control before the BIOS hits the **LCD Refresh RAM**, could set up the Graphic LCD Interface for the particular panel and clear the buffer to ensure that the BIOS doesn't find anything disturbing during the remainder of its scan.

This error was an oversight, pure and simple. I knew (and so did *you*) that the **LCD Refresh RAM** contents survived a reset. It *never* occurred to me that the BIOS might discover an extension in the bits left over from the last message!

If you build anything with dynamic bit patterns in the region where the BIOS expects extensions, take heed. You, too, may spend hard time wondering why your system doesn't boot correctly once in a while.

Release Notes

The files for this chapter check out the Glass Teletype and ANSI code on both the DMF651 and LG64AA44D panels, which are the "best of breed" LCDs in my collection. I wrote the code in Borland C and processed it through Paradigm Locate. Use the appropriate boot sector loader from Chapter 11 to start the files from diskette.