15 Bringing the Graphic LCD Panel to Life

Have you ever repaired something by taking it apart, contemplating the pieces, then putting them back together again? It's hard to believe a simple laying-on of hands can fix an inanimate object, but quite often that's exactly what happens.

Textbooks ascribe such problems to contact corrosion, random glitches, metastability, software errors, and similar maladies. Experienced engineers and technicians know differently. "That widget just wanted some attention," they'll say with a smile and go on to the next problem. To hear them talk, you might almost think the hardware was alive.

So far, we've covered the Graphic LCD Interface's design, hardware, and the test firmware that exercised the circuitry. It's now time for the code that manipulates individual dots on the panel and, as an added bonus, puts your *x*86 target system's extended memory to good use. You'll see several chunks of assembly language here, as we begin writing code where performance matters a bit more.

While the gadgetry isn't quite alive yet, we're getting there…

Putting Life to Work

Graphic LCD panels have a voracious appetite for data: testing the drawing routines means exercising a quarter-million dots in a 640×400 array. Just writing a byte into the RAM and reading it back correctly doesn't mean that the dots appear at the right place on the panel. You must actually *look* at the results to verify that the hardware is working correctly.

But… who wants to watch a test pattern?

Conway's venerable Game of Life is the great-to-the-n^{th} grandfather of today's Artificial Life creations. It produces an easily recognizable set of patterns, exercises the whole LCD panel, is fun to watch, and can be falling-off-a-log easy to code… if performance isn't much of an issue, anyway.

The playing field consists of a rectangular grid of cells, each of which may be either *alive* or *dead*. Ideally, the field should be infinite, but typical implementations surround the visible part of the field with permanently dead cells or join the four edges to simulate a torus. The contents of the initial field uniquely determine the course of the game, making Life a spectator sport rather than an interactive game.

Play proceeds by generation. In principle, all cells change state in one instant at the end of each generation. The state of each cell in the next generation depends on its current state and the state of its eight immediate neighbors, nine cells in all.

A dead cell remains dead unless it has exactly three living neighbors in the current generation, in which case it springs to life in the next generation. A live cell remains alive when exactly two or three of its neighbors are also alive, otherwise it dies of either loneliness or overcrowding. Variations on those simple rules are possible, but tend to produce less interesting patterns.

Listing 1 shows the few lines of code required for this algorithm in its most basic form. A little arithmetic reveals the prodigious computation required for a single generation on 640×400 LCD panel: $9 \times 640 \times 400 = 2.3 \times 10^6$ `LCDGetDot` calls! Because relatively few cells change state after the first few generations, the `LCDSetDot` calls that update the playing field don't influence the overall time.

My oscilloscope reports that the calculations take about 100 µs per cell on a 33 MHz '386SX CPU. Therefore, computing just one generation requires

Listing 1

One of the attractions of Conway's Game of Life is the simplicity of the algorithm: these few lines determine the changed cells in the next generation. Because our graphic LCD panels have a quarter million dots, however, the complete computation can take tens of seconds even on a fast processor. This is fast enough to test our hardware, but you can have fun optimizing the code beyond recognition.

```
for (Row=0; Row<NumRows; Row++) {
   ByteToLEDs(Row);                                  /* show some progress    */
   outpw(SYNC_ADDR,0x80);                            /* for scope timing      */
   for (Col=0; Col<NumCols; Col++) {
      Neighbors = LCDGetDot(Row-1,Col-1) +
         LCDGetDot(Row-1,Col  ) +
         LCDGetDot(Row-1,Col+1) +
         LCDGetDot(Row  ,Col-1) +
         LCDGetDot(Row  ,Col+1) +
         LCDGetDot(Row+1,Col-1) +
         LCDGetDot(Row+1,Col  ) +
         LCDGetDot(Row+1,Col+1);
      if (LCDGetDot(Row,Col)) {
         if ((Neighbors < 2) || (Neighbors > 3)) {
          LCDSetDot(Row,Col,0);
         }
      }
      else {
         if (Neighbors == 3) {
          LCDSetDot(Row,Col,1);
         }
      }
   }
   outpw(SYNC_ADDR,0x00);
}
```

640×400×100 µs = 26 seconds. Watching this process demands nearly the same patience as studying the life cycle of glaciers in real time. Smaller panels run faster, of course, while remaining in the icicle growth competition. Obviously, we have here a program where optimization pays off handsomely.

Michael Abrash presented a Game of Life Optimization Challenge in the June '92 *PC Techniques* magazine, with results appearing in the Dec '93 issue. The two winning entries run at 125 ns/cell (yes, *nanoseconds* per cell) on a 33 MHz '486 (yes, a doorstop system by contemporary desktop PC standards) and their source code is a wonder to behold. Before you start tweaking my simple code, please read those articles… and weep. The contest and results also appear in his *Zen of Code Optimization* book, the contents of which you should commit to memory.

Because I was more interested in LCD panels than the Game of Life, I decided to finesse the problem. I added a few routines that store each successive generation in the target system's extended memory, then play them back as a slide show. You'll get the details after we cover the LCD firmware.

Doing Dots

As you saw in Chapter 13, the Graphic LCD Interface maps the LCD's dots into the **`LCD Refresh RAM`**, a 32 KB chunk of the PC's address space. From the CPU's viewpoint, the LCD panel operates as a typical, albeit relatively slow, RAM on the ISA bus, with the exceedingly useful side effect that its bits are visible to the naked eye. Unfortunately, as we discovered in Chapter 14, each panel has its own dot layout and requires unique code to locate each bit in the RAM.

Finding a particular dot requires two steps: selecting the RAM byte holding the bit, then isolating the bit within that byte. Obviously, both steps depend on the dot's row and column address. To simplify the high level code, rows and columns both start with number 0 in the upper left corner of the panel, even though many LCD panel documents number them differently. Other mappings work equally well: if an upside down, backwards, or rotated mapping suits your application, go for it!

Because of the hardware we're using, the top row of dots on the LCD panel does not come from the first group of bytes in RAM. Although the Graphic LCD Interface produces a **`Frame Sync`** pulse when it resets the **`LCD Address Counter`** to zero, the panel expects **`Frame Sync`** to *follow* the first row. As a result, the dots starting at address 0000 appear on the *second* panel row, not the first.

This was not a problem in Chapter 14, as we were interested in verifying the hardware and showing which RAM addresses appeared on which rows. Now that we must work with specific dot positions, it's time to get precise.

The quick and easy firmware solution for this appears in Listing 2, where I stored the starting address of each row in the `RowStarts` table. That table has one entry for each visible row in the LCD panel: the LG64AA44D table has 400 entries. The first entry points to the bytes at address 199×160 = 31840 decimal = 7c60 hex, while the second entry holds address 0000. The dot drawing routines consult the table to find out where each row starts: the scrambled addresses unscramble the panel layout and put Row 0 at the top of the panel where it belongs.

I create and fill the table as part of an LCD initialization routine which is specific to each panel. Entries 0 and 200 are special-cased at the top of the loop, while the remaining 2×199 entries form a simple ascending sequence. I suspect you could define and fill the table using some assembler macro magic, but this way is easy enough and relatively straightforward to modify.

The `LCDMakeAddr` function in Listing 3 converts a dot's row and column address into a byte address and a bit shift amount. The byte address depends on both the row, which specifies the lookup table entry, and the column, which specifies an additional number of bytes relative to the start of the row.

Listing 2
A lookup table simplifies finding the RAM address of a particular dot by holding the starting address of its row. This code fragment defines and loads the 400 table entries for a Matsushita LG64AA44D 640x400 dot panel. The data for the lower half of the panel is located in the high-order nybble of the same bytes displayed on the upper half, so the second half of the table (Rows 200-399) contains the same addresses as the first half (Rows 0-199).

```
LCDNAME       EQU       "Matsushita EDM LG64AA44D"
NUMROWS       =         400
NUMCOLS       =         640
COLCLOCKS     =         160
COLMODULUS    =         4
TOTALCLOCKS   =         32000
ROWMODULUS    =         NUMROWS/2
FILLVALUE     =         000h

RowStarts     DW        NUMROWS DUP (?)              ; row starting addresses

              MOV       [RowStarts],(ROWMODULUS-1)*COLCLOCKS
              MOV       [RowStarts+2*ROWMODULUS],(ROWMODULUS-1)*COLCLOCKS

              MOV       DI,(OFFSET RowStarts) + 2
              MOV       CX,ROWMODULUS-1
              MOV       AX,0                         ; row 0 = second entry
              PUSH      DS                           ; set up STOSW segment
              POP       ES

@@1:          MOV       [DI+2*ROWMODULUS],AX         ; second half of table
              STOSW                                  ; first half
              ADD       AX,COLCLOCKS                 ; offset of next row
              LOOP      @@1
```

Listing 3
This LG64AA44D panel routine converts a dot's row (in BX) and column (in AX) into the offset of a byte from the start of the buffer (in DI) and the shift amount (in CL) that moves that bit into Bit 0. The high nybble of each byte stores the dots for rows 200 through 399, the low nybble contains dots for rows 0 through 199, and, thus, the shift amount depends on which half of the panel holds the dot.

```
                PROC    LCDMakeAddr

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

                DIV     [BYTE ColModulus]
                XOR     AH,03h              ; flip shift direction
                MOV     CL,AH               ; save remainder for shifting
                MOV     AH,0
                ADD     DI,AX               ; DI points to the byte

                CMP     BX,2*ROWMODULUS     ; rows >=200 use high nybble
                JB      @@1
                ADD     CL,4                ; ... so shift into it
@@1:

                RET

                ENDP    LCDMakeAddr
```

The bit shift value in `CL` moves the selected bit to the low-order bit or vice versa. Depending on which panel we're using, the shift amount may also depend on the row, the column, or both, because these LCD panels pack several widely separated dots together in each **`LCD Refresh RAM`** byte.

Homework assignment: step through the code samples with a pencil and paper to see how the algorithm works. Draw a picture of the LCD panel's bit layout, indicate the bit addresses at the start of each row, then verify that the code really does produce the right answers. You won't believe it until you do…

Listing 4 pulls all of this together to write a single dot. As you saw in Listing 1, the `main()` C routine calls `LCDSetDot` directly, which means that routine's register and stack usage *must* match C's expectations. Interfacing assembler with C used to be fiendishly tricky, but current PC assemblers include several high level directives to ease the task. I won't go into the details here, other than to say it's *much* easier than counting bytes and tweaking registers on your own.

Recall that `LCDMakeAddr` returns a byte offset in `DI` rather than a complete `seg:off` address. `LCDSetDot` adds the far pointer in `pNewBuff` to `DI` to create the final address in `ES:DI`. You can aim `LCDSetDot` at the actual **`LCD`**

Listing 4
This routine sets or clears a dot on the LG64AA44D LCD panel. The global variable pNewBuff contains a far pointer to the target buffer, which may be the LCD panel or a chunk of system RAM. Depending on your application, you may want to include the target buffer address as one of the function parameters. This code can be called directly from the main C program, because the first four lines handle the inter-language register and stack conversions.

```
                PUBLIC    C LCDSetDot
                PROC      LCDSetDot
                ARG       Row:WORD,Col:WORD,Value:WORD
                USES      ES,DI

                MOV       AX,[Col]
                MOV       BX,[Row]
                CALL      LCDMakeAddr

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

                MOV       BX,0101h            ; set up bit mask
                SHL       BX,CL
                NOT       BL                  ;  BL = AND mask, BH = OR mask

                TEST      [Value],0001h       ; what do we want?
                JNZ       @@2                 ; nonzero means set the bit
                AND       [ES:DI],BL          ;  ... clear it
                JMP       SHORT @@3
@@2:
                OR        [ES:DI],BH          ;  ... set it

@@3:
                RET

                ENDP      LCDSetDot
```

`Refresh RAM` addresses or, for more speed, at a separate working buffer in system RAM, just by changing `pNewBuff`.

Normally, you would include the buffer base address in `LCDSetDot`'s arguments. I used a global variable to reduce the amount of stack shuffling, but I agree that global variables can cause problems in a real application. As always, use this code as the basis for your own efforts, rather than The Final Word on Programming Style.

The `BX` register holds the bit masks that set or clear the selected bit. Although only one instruction references the target byte in each pass through Listing 4, the CPU actually makes two memory accesses: one to read the current contents and another to write the updated dots back into the same byte. This instruction runs as fast as

the hardware can do it, but the PC Compatibility Barnacles make the **LCD Refresh RAM** on the ISA bus achingly slow compared to system board DRAM.

Listing 5 shows how `LCDGetDot` tests an LCD bit. In this routine, the global far pointer `pOldBuff` contains the buffer's RAM address. The shift amount in `CL` now moves the selected bit into the LSB of `AX`, where we mask it to produce a Boolean return value for the calling routine.

`LCDGetDot` and `LCDSetDot` use two different global pointers, because the Game of Life's playing field must remain unchanged until *after* the current generation tests all the old cells. I aimed `pNewBuff` at the **LCD Refresh RAM** buffer to allow us to see the dots appear, but you could put them in system RAM. In that case, just copy all of them to the **LCD Refresh RAM** at the end of each generation to reveal the new field in one blink.

Listing 5

This routine tests a dot on the LG64AA44D LCD panel and returns its state as a C Boolean value. If the row or column lies outside the LCD's boundaries, the return value is always zero. While this simplifies the higher level code for the Game of Life, it may not be the right action for all programs!

```
                PUBLIC    C LCDGetDot
                PROC      LCDGetDot
                ARG       Row:WORD,Col:WORD
                USES      ES,DI

                XOR       AX,AX               ; set up zero return value

                MOV       BX,[Col]            ; verify column
                TEST      BH,80h              ; negative?
                JNZ       @@Done              ;  ... yes
                CMP       BX,NUMCOLS          ; too big?
                JAE       @@Done              ;  ... yes

                MOV       BX,[Row]            ; verify row
                TEST      BH,80h              ; negative?
                JNZ       @@Done              ;  ... yes
                CMP       BX,NUMROWS          ; too big?
                JAE       @@Done              ;  ... yes

                MOV       AX,[Col]            ; set up column number
                CALL      LCDMakeAddr

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

                MOV       AL,[ES:DI]          ;  ... fetch the byte
                SHR       AL,CL               ; skid our bit into LSB
                AND       AX,0001h            ;  ... isolate it
@@Done:
                RET

                ENDP      LCDGetDot
```

The main loop calls `LCDSetDot` only when a cell changes, which means the new field must contain all the unchanged cells at the start of each generation. The C `memcpy()` library function copies the entire 32 KB buffer from the **`LCD Refresh RAM`** into system RAM at the end of each generation. Compared to the time required to compute a generation, `memcpy()` runs in essentially zero time.

Variations on a Theme

Even through the LG64AA44D panel has a moderately complex dot layout, the code remains fairly simple. Rather than list the routines for the 640×200 DMF651 and the TLY-365-121 panels here, check the source code files for the full details. I decided not to do the Sharp LM64015T 640×400 panel, because I still don't have a reliable source for its backlight inverters. If you can get one, have at it!

The 480×128 LM215 merits some discussion, if only because it has such a peculiar layout. Each of the four bits in the low nybble drives a separate quadrant: in effect, the panel has four 240×64 subpanels. It uses a 960 ns **`Dot Clock`**, so the firmware

Listing 6
The LM215 panel has more complex addressing, because it is actually four 240x64 panels butted together on one piece of glass. Both the bit shift amount and the byte offset depend on the dot's quadrant. The SETAE instructions, peculiar to the '386 and higher CPUs, set CL to 1 if the preceding CMP was "above or equal" and to 0 otherwise.

```
LCDNAME       EQU       "Hitachi LM215 (960 ns, set jumpers!)"
NUMROWS       =         128
NUMCOLS       =         480
TOTALCLOCKS   =         30720
ROWCLOCKS     =         480                 ; every other byte in buffer!
COLMODULUS    =         NUMCOLS/2
ROWMODULUS    =         NUMROWS/2
FILLVALUE     =         000h

              PROC      LCDMakeAddr

              CMP       AX,COLMODULUS       ; column
              SETAE     CL                  ; CL = 1 if in right half
              ADD       CL,CL               ;  ... make it 2
              CMP       BX,ROWMODULUS       ; upper or lower half?
              SETAE     CH                  ; CH = 1 if in lower half
              ADD       CL,CH               ; get total shift amount

              ADD       BX,BX               ; make word table index
              MOV       DI,[RowStarts+BX]   ;  ... to get row start

              DIV       [BYTE ColModulus]   ; get column offset
              MOV       AL,AH               ; remainder = col offset
              XOR       AH,AH               ; always less than 256...
              ADD       AX,AX               ; times 2 for addressing
              ADD       DI,AX               ; set up complete address

              RET
              ENDP      LCDMakeAddr
```

must also duplicate the dots in successive even- and odd-addressed **LCD Refresh RAM** bytes to present stable data to the panel.

The LM215's row address table has 128 entries, corresponding to the 128 visible rows. The code I used to fill it resembles that shown in Listing 2. I put only the even addresses in the table and modified the `LCDGetDot` and `LCDSetDot` functions to generate the corresponding odd addresses.

Listing 6 shows the LM215 `LCDMakeAddr` function. The bit shift amount depends on the dot's quadrant, a perfect application for the '386 `SETAE` instruction that generates a Boolean value from the flags set by `CMP`. The byte offset added to the table entry is the same for the first and last 240 columns in each row. Because `SETAE` doesn't exist on CPUs prior to the '386, you *must* tweak the code if you have an 8088 or 80286 target system.

Listing 7
The LM215 requires a 960 ns Dot Clock, so identical data must appear in successive even- and odd-addressed bytes to provide stable signals to the panel. The remainder of this code resembles that used for other, more normal, LCD panels, because LCDMakeAddr hides the addressing differences.

```
                PUBLIC    C LCDSetDot
                PROC      LCDSetDot
                ARG       Row:WORD,Col:WORD,Value:WORD
                USES      ES,DI

                MOV       AX,[Col]
                MOV       BX,[Row]
                CALL      LCDMakeAddr

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

                MOV       BX,0101h            ; set up bit mask
                SHL       BX,CL
                NOT       BL                  ;  BL = AND mask, BH = OR mask

                TEST      [Value],0001h       ; what do we want?
                JNZ       @@2                 ; nonzero means set the bit
                AND       [ES:DI],BL          ;  ... clear it
                AND       [ES:DI+1],BL        ;  ... do the odd byte, too
                JMP       SHORT @@3

@@2:
                OR        [ES:DI],BH          ;  ... set it
                OR        [ES:DI+1],BH        ;  ... do the odd byte, too

@@3:
                RET

                ENDP      LCDSetDot
```

The LM215 `LCDSetDot` function in Listing 7 updates both bytes with the new dot value. Apart from that, it resembles the code for the other panels because `LCDMakeAddr` handles the panel's peculiar bit and byte addressing.

The LCD initialization routines for all these panels disable blinking by gating a constant zero to the **`LCD Data Multiplexer`**. While you may have good reasons for showing blinking dots, the Game of Life doesn't need any. I leave that as an exercise for the interested reader (yes, *you*).

Stashing Slides

I used the Game of Life because its outputs make a far more interesting test pattern than a bland bit-by-bit diagnostic program. Unfortunately, waiting half a minute for each update puts a strain on even the most placid attention span. Rather than get involved with code optimization, I decided to store successive generations in RAM and play them back quickly. While the slide show repeats after a while, that allows you to study problems as they crop up over and over again… it's a perfect test-pattern application.

So far in this book, our *x*86 target CPUs have been running in real mode, which allows only 20 address bits and limits RAM addresses to 1 MB. The PC's memory addressing restrictions, which we discussed in Chapters 6 and 7, limit contiguous RAM to the 640 KB starting at physical address 00000. After loading a program, allocating a few work buffers, and leaving room for a decent `FAR` heap area, the system may have only a few hundred KB available below the video buffers.

In contrast, when a '386 or higher CPU runs in protected mode, it can use all 32 address bits to reach 4 GB of memory. Even the lowly '386SX chip has 24 address pins and can access 16 MB, still significantly more than a paltry 640 KB. In most PC systems, the first megabyte of the protected mode address space remains devoted to the familiar real mode RAM and I/O devices, with the remaining storage appearing as a contiguous expanse starting at the 1 MB line.

This RAM area, generally called *extended memory*, differs from *expanded memory* that can be page mapped into some part of the lower 1 MB. The (now largely irrelevant) LIM 4.0 spec defined the operation of a standard DOS Expanded Memory Manager that controlled actual memory mapping hardware. The `EMM386` DOS device driver controls extended memory and can simulate expanded memory with no additional hardware. Refer back to Chapter 6 for a discussion of how that affected our ability to locate memory on the ISA bus.

Although the CPU must run in protected mode to access extended memory, we don't need a complete protected mode program to use that RAM. The system

BIOS includes functions that copy blocks of data in extended memory to and from the 1 MB available to real-mode programs, allowing us to create the data as usual and store it through a BIOS call.

The process begins by finding out how much extended memory resides on the target system. Listing 8 shows the code to access BIOS function `INT 15h, AH=88h`, which returns the extended memory size in kilobytes in register `AX`. Because `AX` holds only 16 bits, this function can handle a mere 64 MB of RAM… enough for an old '386SX, but painfully cramped in the days of Pentium Pro CPUs with nearly 1 GB of RAM on their system boards.

The various references disagree as to whether the `INT 15h, AH=88h` function clears the `Carry` flag to indicate success. In my test systems, `Carry` remains set even though `AX` contains the correct value, so I added a test to weed out some obviously bogus return values. If it misbehaves on your system, write a few testcases and see how your BIOS behaves.

The firmware turns bit 0 of the printer port at `SYNC_ADDR` (typically, LPT1) ON before invoking the software interrupt and OFF when it returns. If that bit stays ON, you know your system has problems. This will *certainly* happen on 8088 systems and will probably happen on oddball '286 systems, because the BIOS functions aren't quite compatible.

Function `INT 15h, AH=88h` reads the extended memory size from the battery backed RAM in the CMOS Real-Time Clock chip, where, on some systems, the

Listing 8

BIOS function Int 15h, AH=88h returns the extended memory size in KB. This memory starts just beyond the 1 MB of memory addressable in real mode. Because AX has only 16 bits, the function can report only the first 64 MB of extended memory!

```
regs.h.ah = 0x88;
outp(SYNC_ADDR,0x01);                                   /* mark the query        */
int86(0x15,&regs,&regs);
outp(SYNC_ADDR,0x00);

XMemSize = regs.x.ax;
printf(" BIOS reports %u KB\n",XMemSize);

if ((XMemSize >> 8) >= 0x0080) {
   puts(" ... on a small PC you can't have that much, so it's ignored");
   XMemSize = 0;
}

XMemLimit = (long)(XMemSize / FRAMESIZE);           /* truncate limit to a   */
XMemLimit *= (long)FRAMESIZE * 1024L;               /*  multiple of the slide*/
XMemLimit += ONE_MEG;                               /* above the line        */
printf(" Room for %u frames of %u KB each between %08lX and %08lX\n",
   XMemSize/FRAMESIZE,FRAMESIZE,ONE_MEG,XMemLimit-1);
```

BIOS stored it during the power-on tests after each reset. Other BIOSes store this value as part of a manual setup and complain if it doesn't match the actual amount of RAM found during POST. In any event, the return value should be exactly 1 MB less than the total amount of RAM installed in your system.

BIOS function `INT 15h, AH=87h` copies a block of data between any two locations in memory. The process looks straightforward: enter the source and destination addresses in a table in RAM, specify the number of 16-bit words to move in `CX`, aim `ES:SI` at the table, and issue the software interrupt. However, the references disagree on exactly what the table should contain in addition to the addresses, if anything.

It turns out the table actually forms the GDT (Global Descriptor Table) used when the CPU enters protected mode. Listing 9 shows the structure of the GDT entries: even when the code runs on a 32-bit '386 or higher CPU, the BIOS expects a 16-bit, 80286-style GDT for compatibility with the Original PC AT. The two addresses form part of a pair of segment descriptors that the CPU uses to access the data blocks in extended memory.

Listing 10 shows how I copied the Life fields into extended memory. The GDT requires memory addresses in 24-bit linear format, not the `seg:off` pairs we all love to hate. I filled in the GDT entries holding the segment length and access flag bits, even though some of the references imply the BIOS will do this automatically.

The code copies the entire 32 KB block of storage in order to make the extended memory addresses easy to decode by eyeball inspection. If you apply data

Listing 9
BIOS Int 15, AH=87h copies memory between any two addresses in RAM, specified as 24-bit linear values in a Global Descriptor Table, rather than the familiar seg:off pairs used below 1 MB. The GDT, made up of an array of the structures shown here, defines the CPU's access to memory in protected mode.

```
#pragma option -a-                    /* structure must be packed          */
typedef struct {
   WORD SegLimit;                     /* size of segment in bytes          */
   WORD SegBaseLow;                   /* low bytes of linear address       */
   BYTE SegBaseHigh;                  /* high byte of linear address       */
   BYTE SegFlags;                     /* 9B=code, 93=data, 91=RO           */
   WORD Reserved;                     /* reserved, must be zero            */
} GDT_286;
#pragma option -a.                    /* restore default setting           */

#define MK_LINEAR(fp) (((long)FP_SEG(fp) << 4) + (long)FP_OFF(fp))

GDT_286 MoveGDT[6];                   /* GDT for BIOS Move Block fn        */
```

Listing 10
In addition to the (obviously) necessary source and destination addresses in the GDT, this code fills in the segment length and access flags even though they're not required on my target systems. Bit 1 on the printer port marks the CPU's journey through protected mode; if that LED remains lit, your CPU fell off the tracks. The code to copy data back from extended memory is similar: just interchange the addresses.

```
memset(MoveGDT,0,sizeof(MoveGDT));              /* clear the GDT         */

MoveGDT[2].SegLimit = FRAMESIZE*1024;
MoveGDT[2].SegFlags = 0x93;
MoveGDT[2].SegBaseLow = (WORD)MK_LINEAR(pOldBuff);
MoveGDT[2].SegBaseHigh = (BYTE)(MK_LINEAR(pOldBuff) >> 16);
MoveGDT[3].SegLimit = FRAMESIZE*1024;
MoveGDT[3].SegFlags = 0x93;
MoveGDT[3].SegBaseLow = (WORD)XMemAddr;         /* target                */
MoveGDT[3].SegBaseHigh = (BYTE)(XMemAddr >> 16);

segread(&sregs);                                /* get current seg regs  */
regs.h.ah = 0x87;                               /* move block            */
regs.x.cx = FRAMESIZE*(1024/2);                 /* number of words       */
regs.x.si = FP_OFF(MoveGDT);                    /* offset of GDT         */
sregs.es = FP_SEG(MoveGDT);                     /*  ... segment          */

outp(SYNC_ADDR,0x02);                           /* mark the move         */
int86x(0x15,&regs,&regs,&sregs);                /* shazam!               */
outp(SYNC_ADDR,0x00);

printf(" stored at %08lX",XMemAddr);

XMemAddr += (long)FRAMESIZE * 1024L;            /* step to next block    */
```

compression before storing the frames, remember to tweak CX and step the addresses by the appropriate amount.

The two `outpw()` functions bracket the CPU's journey through protected mode by blinking bit 1 on the printer port: if that bit stays ON, you know your system got lost in hyperspace. My 33 MHz '386SX system takes about 32 ms to transfer a 32 KB block, about 1 MB/s. Because the transfer stays entirely within the system board RAM, the usual ISA bus slowdown doesn't come into play. You can easily see that switching to and from protected mode adds considerable overhead! A little experimentation with different buffer locations should reveal the size of the penalty. Hint: put *both* buffers in system memory and time it again.

Because the BIOS doesn't have access to any interrupt handlers in protected mode, it must disable all hardware interrupts during the entire transfer. If your code must receive high speed serial data or process other closely spaced interrupts, this function simply will not work for you. Also, the references lists several possible error return codes that I simply ignore here. You must deal with all those issues in a real application, of course.

After filling extended memory with Life generations, the code once again invokes `INT 15h, AH=87h` to copy the patterns back into memory below 1 MB. One of the FDB's DIP switches selects either `memcpy()` or a nested pair of C loops for the subsequent copy into the **`LCD Refresh RAM`**, allowing you to see the effects. A hardcoded 500 ms delay between each slide also gives you time to think about what's happening and recognize any problems.

One megabyte of extended memory holds 32 generations of 32 KB each, about 16 seconds. An 8 MB system, with 7 MB above the 1 MB line, stores 224 generations and presents a two-minute slice of Life. The low-order byte of the DIP switches on the Firmware Development Board sets the initial random number seed and, thus, selects one of 255 different slide shows. You can modify the code to pick more bits from the DIP switches or use an entirely different random number generator.

Because 200-line panels use only the low-order nybble of each byte, you can obviously store twice as many generations with a no-brainer data compression scheme. The Life field has many dead cells after the first few generations and you can probably pick up another 50% by run length encoding what's left… have fun!

Don't be surprised to find yourself rooting for those clumps of dots as they expand, contract, and consume each other. Life is like that…

Release Notes

The sample program files include the source code and `BIN` files for the four panels I mentioned in the text, written in Borland C and processed through Paradigm's `Locate` utility. Copy the `BIN` files to a diskette with the boot loader from Chapter 11 and bring some life to the LCD panel on your target system.

The code unique to each panel resides in an assembler file bearing the panel's name. The `MakeFile` produces a similarly named `BIN` file on drive A, so have a diskette with the boot loader ready when you recompile the code. If you add another type of panel, use an existing file as a template, update the `MakeFile`, and you're all set.

You also get my version of Paradigm's `Console.C` file. I replaced their demo code with BIOS serial functions to support simple, polled console I/O. If you're using `Locate`, you must also set up the `FARHEAP` constant that allocates a decent sized area for the program's work buffers. I used 0x8000, but you may find a smaller value works fine.

Homework: you can build some useful graphic code atop the dot drawing routines. For vector graphics, add line and arc drawing routines. For bitmaps, you'll need block move and copy operations that properly handle the scattered bits.