# Chapter 47

# Mode X: 256-Color VGA Magic

# Chapter 47

## Introducing the VGA's Undocumented "Animation-Optimal" Mode

At a book signing for my book *Zen of Code Optimization*, an attractive young woman came up to me, holding my book, and said, "You're Michael Abrash, aren't you?" I confessed that I was, prepared to respond in an appropriately modest yet proud way to the compliments I was sure would follow. (It was my own book signing, after all.) It didn't work out quite that way, though. The first thing out of her mouth was:

"'Mode X' is a stupid name for a graphics mode." As my jaw started to drop, she added, "And you didn't invent the mode, either. My husband did it before you did."

And they say there are no groupies in programming!

Well. I never claimed that I invented the mode (which is a 320×240 256-color mode with some very special properties, as we'll see shortly). I did discover it independently, but so did other people in the game business, some of them no doubt before I did. The difference is that all those other people held onto this powerful mode as a trade secret, while I didn't; instead, I spread the word as broadly as I could in my column in *Dr. Dobb's Journal*, on the theory that the more people knew about this mode, the more valuable it would be. And I succeeded, as evidenced by the fact that this now widely-used mode is universally known by the name I gave it in *DDJ*, "Mode X." Neither do I think that's a bad name; it's short, catchy, and easy to remember, and it befits the mystery status of this mode, which was omitted entirely from IBM's documentation of the VGA.

In fact, when all is said and done, Mode X is one of my favorite accomplishments. I remember reading that Charles Schultz, creator of "Peanuts," was particularly proud of having introduced the phrase "security blanket" to the English language. I feel much the same way about Mode X; it's now a firmly entrenched part of the computer lexicon, and how often do any of us get a chance to do that? And that's not to mention all the excellent games that would not have been as good without Mode X.

So, in the end, I'm thoroughly pleased with Mode X; the world is a better place for it, even if it did cost me my one potential female fan. (Contrary to popular belief, the lives of computer columnists and rock stars are not, repeat, *not*, all that similar.) This and the following two chapters are based on the *DDJ* columns that started it all back in 1991, three columns that generated a tremendous amount of interest and spawned a ton of games, and about which I still regularly get letters and e-mail. Ladies and gentlemen, I give you...Mode X.

## What Makes Mode X Special?

Consider the strange case of the VGA's 320×240 256-color mode—Mode X—which is undeniably complex to program and isn't even documented by IBM—but which is, nonetheless, perhaps the single best mode the VGA has to offer, especially for animation.

We've seen the VGA's undocumented 256-color modes, in Chapters 31 and 32, but now it's time to delve into the wonders of Mode X itself. (Most of the performance tips I'll discuss for this mode also apply to the other non-standard 256-color modes, however.) Five features set Mode X apart from other VGA modes. First, it has a 1:1 aspect ratio, resulting in equal pixel spacing horizontally and vertically (that is, square pixels). Square pixels make for the most attractive displays, and avoid considerable programming effort that would otherwise be necessary to adjust graphics primitives and images to match the screen's pixel spacing. (For example, with square pixels, a circle can be drawn as a circle; otherwise, it must be drawn as an ellipse that corrects for the aspect ratio—a slower and considerably more complicated process.) In contrast, mode 13H, the only documented 256-color mode, provides a nonsquare 320×200 resolution.

Second, Mode X allows page flipping, a prerequisite for the smoothest possible animation. Mode 13H does not allow page flipping, nor does mode 12H, the VGA's high-resolution 640×480 16-color mode.

Third, Mode X allows the VGA's plane-oriented hardware to be used to process pixels in parallel, improving performance by up to four times over mode 13H.

Fourth, like mode 13H but unlike all other VGA modes, Mode X is a byte-per-pixel mode (each pixel is controlled by one byte in display memory), eliminating the slow read-before-write and bit-masking operations often required in 16-color modes, where each byte of display memory represents more than a single pixel. In addition to cutting the number of memory accesses in half, this is important because the 486/ Pentium write FIFO and the memory caching schemes used by many VGA clones speed up writes more than reads.

Fifth, unlike mode 13H, Mode X has plenty of offscreen memory free for image storage. This is particularly effective in conjunction with the use of the VGA's latches; together, the latches and the off-screen memory allow images to be copied to the screen four pixels at a time.

There's a sixth feature of Mode X that's *not* so terrific: It's hard to program efficiently. As Chapters 23 through 30 of this book demonstrates, 16-color VGA programming can be demanding. Mode X is often as demanding as 16-color programming, and operates by a set of rules that turns everything you've learned in 16-color mode sideways. Programming Mode X is nothing like programming the nice, flat bitmap of mode 13H, or, for that matter, the flat, linear (albeit banked) bitmap used by 256-color SuperVGA modes. (I't's important to remember that Mode X works on *all* VGAs, not just SuperVGAs.) Many programmers I talk to love the flat bitmap model, and think that it's the ideal organization for display memory because it's so straightforward to program. Here, however, the complexity of Mode X is opportunity—opportunity for the best combination of performance and appearance the VGA has to offer. If you do 256-color programming, and especially if you use animation, you're missing the boat if you're not using Mode X.

Although some developers have taken advantage of Mode X, its use is certainly not universal, being entirely undocumented; only an experienced VGA programmer would have the slightest inkling that it even exists, and figuring out how to make it perform beyond the write pixel/read pixel level is no mean feat. Little other than my *DDJ* columns has been published about it, although John Bridges has widely distributed his code for a number of undocumented 256-color resolutions, and I'd like to acknowledge the influence of his code on the mode set routine presented in this chapter.

Given the tremendous advantages of Mode X over the documented mode 13H, I'd very much like to get it into the hands of as many developers as possible, so I'm going to spend the next few chapters exploring this odd but worthy mode. I'll provide mode set code, delineate the bitmap organization, and show how the basic write pixel and read pixel operations work. Then, I'll move on to the magic stuff: rectangle fills, screen clears, scrolls, image copies, pixel inversion, and, yes, polygon fills (just a different driver for the polygon code), all blurry fast; hardware raster ops; and page flipping. In the end, I'll build a working animation program that shows many of the features of Mode X in action.

The mode set code is the logical place to begin.

## Selecting 320x240 256-Color Mode

We could, if we wished, write our own mode set code for Mode X from scratch—but why bother? Instead, we'll let the BIOS do most of the work by having it set up mode 13H, which we'll then turn into Mode X by changing a few registers. Listing 47.1 does exactly that.

The code in Listing 47.1 has been around for some time, and the very first version had a bug that serves up an interesting lesson. The original *DDJ* version made images roll on IBM's fixed-frequency VGA monitors, a problem that didn't come to my attention until the code was in print and shipped to 100,000 readers.

The bug came about this way: The code I modified to make the Mode X mode set code used the VGA's 28-MHz clock. Mode X should have used the 25-MHz clock, a simple matter of setting bit 2 of the Miscellaneous Output register (3C2H) to 0 instead of 1.

Alas, I neglected to change that single bit, so frames were drawn at a faster rate than they should have been; however, both of my monitors are multifrequency types, and they automatically compensated for the faster frame rate. Consequently, my clock-selection bug was invisible and innocuous—until it was distributed broadly and everybody started banging on it.

IBM makes only fixed-frequency VGA monitors, which require very specific frame rates; if they don't get what you've told them to expect, the image rolls. The corrected version is the one shown here as Listing 47.1; it does select the 25-MHz clock, and works just fine on fixed-frequency monitors.

Why didn't I catch this bug? Neither I nor a single one of my testers had a fixed-frequency monitor! This nicely illustrates how difficult it is these days to test code in all the PC-compatible environments in which it might run. The problem is particularly severe for small developers, who can't afford to buy every model of every hardware component from every manufacturer; just imagine trying to test network-aware software in all possible configurations!

When people ask why software isn't bulletproof; why it crashes or doesn't coexist with certain programs; why PC clones aren't always compatible; why, in short, the myriad irritations of using a PC exist—this is a big part of the reason. I guess that's just the price we pay for the unfettered creativity and vast choice of the PC market.

### LISTING 47.1 L47-1.ASM

```
; Mode X (320x240, 256 colors) mode set routine. Works on all VGAs.
; ****************************************************************
; * Revised 6/19/91 to select correct clock; fixes vertical roll *
; * problems on fixed-frequency (IBM 851X-type) monitors.        *
; ****************************************************************
; C near-callable as:
;       void Set320x240Mode(void);
; Tested with TASM
; Modified from public-domain mode set code by John Bridges.

SC_INDEX        equ     03c4h   ;Sequence Controller Index
CRTC_INDEX      equ     03d4h   ;CRT Controller Index
MISC_OUTPUT     equ     03c2h   ;Miscellaneous Output register
SCREEN_SEG      equ     0a000h  ;segment of display memory in mode X

        .model  small
        .data
```

```
; Index/data pairs for CRT Controller registers that differ between
; mode 13h and mode X.
CRTParms label  word
        dw      00d06h  ;vertical total
        dw      03e07h  ;overflow (bit 8 of vertical counts)
        dw      04109h  ;cell height (2 to double-scan)
        dw      0ea10h  ;v sync start
        dw      0ac11h  ;v sync end and protect cr0-cr7
        dw      0df12h  ;vertical displayed
        dw      00014h  ;turn off dword mode
        dw      0e715h  ;v blank start
        dw      00616h  ;v blank end
        dw      0e317h  ;turn on byte mode
CRT_PARM_LENGTH equ     (($-CRTParms)/2)

        .code
        public  _Set320x240Mode
_Set320x240Mode proc    near
        push    bp      ;preserve caller's stack frame
        push    si      ;preserve C register vars
        push    di      ; (don't count on BIOS preserving anything)

        mov     ax,13h  ;let the BIOS set standard 256-color
        int     10h     ; mode (320x200 linear)

        mov     dx,SC_INDEX
        mov     ax,0604h
        out     dx,ax   ;disable chain4 mode
        mov     ax,0100h
        out     dx,ax   ;synchronous reset while setting Misc Output
                        ; for safety, even though clock unchanged
        mov     dx,MISC_OUTPUT
        mov     al,0e3h
        out     dx,al   ;select 25 MHz dot clock & 60 Hz scanning rate

        mov     dx,SC_INDEX
        mov     ax,0300h
        out     dx,ax   ;undo reset (restart sequencer)

        mov     dx,CRTC_INDEX ;reprogram the CRT Controller
        mov     al,11h  ;VSync End reg contains register write
        out     dx,al   ; protect bit
        inc     dx      ;CRT Controller Data register
        in      al,dx   ;get current VSync End register setting
        and     al,7fh  ;remove write protect on various
        out     dx,al   ; CRTC registers
        dec     dx      ;CRT Controller Index
        cld
        mov     si,offset CRTParms ;point to CRT parameter table
        mov     cx,CRT_PARM_LENGTH ;# of table entries
SetCRTParmsLoop:
        lodsw           ;get the next CRT Index/Data pair
        out     dx,ax   ;set the next CRT Index/Data pair
        loop    SetCRTParmsLoop

        mov     dx,SC_INDEX
        mov     ax,0f02h
        out     dx,ax   ;enable writes to all four planes
        mov     ax,SCREEN_SEG ;now clear all display memory, 8 pixels
        mov     es,ax         ; at a time
```

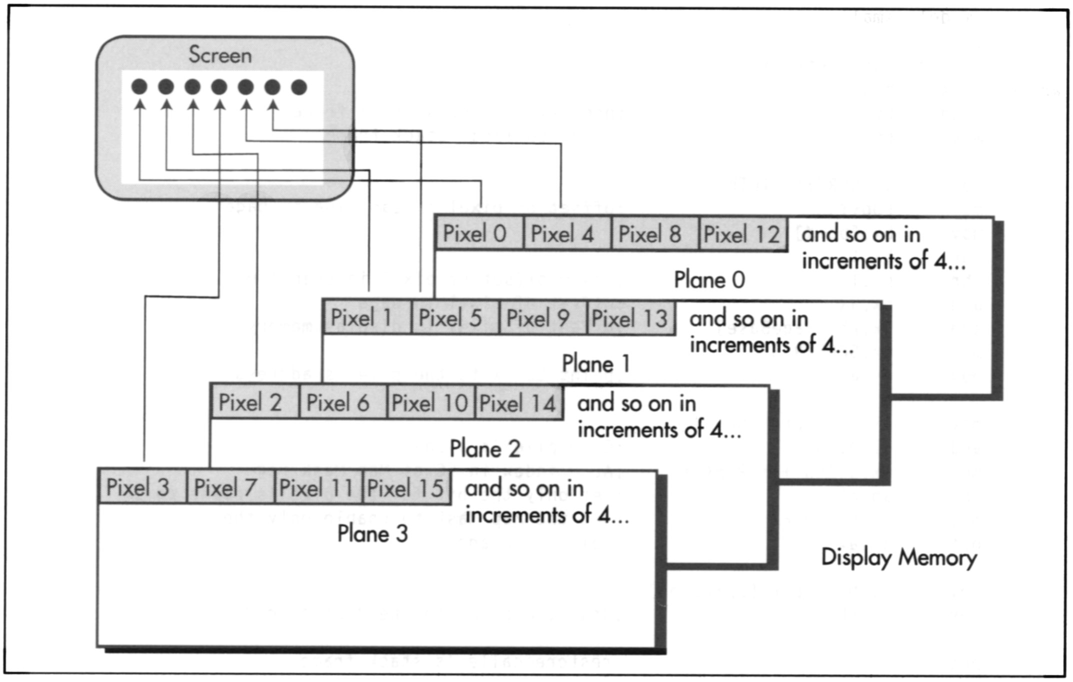
```
        sub     di,di   ;point ES:DI to display memory
        sub     ax,ax   ;clear to zero-value pixels
        mov     cx,8000h ;# of words in display memory
        rep     stosw   ;clear all of display memory

        pop     di      ;restore C register vars
        pop     si
        pop     bp      ;restore caller's stack frame
        ret
_Set320x240Mode endp
        end
```

After setting up mode 13H, Listing 47.1 alters the vertical counts and timings to select 480 visible scan lines. (There's no need to alter any horizontal values, because mode 13H and Mode X both have 320-pixel horizontal resolutions.) The Maximum Scan Line register is programmed to double scan each line (that is, repeat each scan line twice), however, so we get an effective vertical resolution of 240 scan lines. It is, in fact, possible to get 400 or 480 independent scan lines in 256-color mode, as discussed in Chapter 31 and 32; however, 400-scan-line modes lack square pixels and can't support simultaneous off-screen memory and page flipping. Furthermore, 480-scan-line modes lack page flipping altogether, due to memory constraints.

At the same time, Listing 47.1 programs the VGA's bitmap to a planar organization that is similar to that used by the 16-color modes, and utterly different from the linear bitmap of mode 13H. The bizarre bitmap organization of Mode X is shown in Figure 47.1. The first pixel (the pixel at the upper left corner of the screen) is controlled by the byte at offset 0 in plane 0. (The one thing that Mode X blessedly has in common with mode 13H is that each pixel is controlled by a single byte, eliminating the need to mask out individual bits of display memory.) The second pixel, immediately to the right of the first pixel, is controlled by the byte at offset 0 in plane 1. The third pixel comes from offset 0 in plane 2, and the fourth pixel from offset 0 in plane 3. Then, the fifth pixel is controlled by the byte at offset 1 in plane 0, and that cycle continues, with each group of four pixels spread across the four planes at the same address. The offset M of pixel N in display memory is M = N/4, and the plane P of pixel N is P = N mod 4. For display memory writes, the plane is selected by setting bit P of the Map Mask register (Sequence Controller register 2) to 1 and all other bits to 0; for display memory reads, the plane is selected by setting the Read Map register (Graphics Controller register 4) to P.

It goes without saying that this is one ugly bitmap organization, requiring a lot of overhead to manipulate a single pixel. The write pixel code shown in Listing 47.2 must determine the appropriate plane and perform a 16-bit **OUT** to select that plane for each pixel written, and likewise for the read pixel code shown in Listing 47.3. Calculating and mapping in a plane once for each pixel written is scarcely a recipe for performance.

That's all right, though, because most graphics software spends little time drawing individual pixels. I've provided the write and read pixel routines as basic primitives,




*Mode X display memory organization.*
**Figure 47.1**

and so you'll understand how the bitmap is organized, but the building blocks of high-performance graphics software are fills, copies, and bitblts, and it's there that Mode X shines.

### LISTING 47.2 L47-2.ASM

```
; Mode X (320x240, 256 colors) write pixel routine. Works on all VGAs.
; No clipping is performed.
; C near-callable as:
;
;    void WritePixelX(int X, int Y, unsigned int PageBase, int Color);

SC_INDEX        equu    03c4h     ;Sequence Controller Index
MAP_MASK        equ     02h       ;index in SC of Map Mask register
SCREEN_SEG      equ     0a000h    ;segment of display memory in mode X
SCREEN_WIDTH    equ     80        ;width of screen in bytes from one scan line
                                  ; to the next

parms   struc
        dw      2 dup (?)         ;pushed BP and return address
X       dw      ?                 ;X coordinate of pixel to draw
Y       dw      ?                 ;Y coordinate of pixel to draw
PageBase dw     ?                 ;base offset in display memory of page in
                                  ; which to draw pixel
Color   dw      ?                 ;color in which to draw pixel
parms   ends
```

```
        .model  small
        .code
        public  _WritePixelX
_WritePixelX    proc    near
        push    bp                      ;preserve caller's stack frame
        mov     bp,sp                   ;point to local stack frame

        mov     ax,SCREEN_WIDTH
        mul     [bp+Y]                  ;offset of pixel's scan line in page
        mov     bx,[bp+X]
        shr     bx,1
        shr     bx,1                    ;X/4 = offset of pixel in scan line
        add     bx,ax                   ;offset of pixel in page
        add     bx,[bp+PageBase]        ;offset of pixel in display memory
        mov     ax,SCREEN_SEG
        mov     es,ax                   ;point ES:BX to the pixel's address

        mov     cl,byte ptr [bp+X]
        and     cl,011b                 ;CL = pixel's plane
        mov     ax,0100h + MAP_MASK     ;AL = index in SC of Map Mask reg
        shl     ah,cl                   ;set only the bit for the pixel's plane to 1
        mov     dx,SC_INDEX             ;set the Map Mask to enable only the
        out     dx,ax                   ; pixel's plane

        mov     al,byte ptr [bp+Color]
        mov     es:[bx],al              ;draw the pixel in the desired color

        pop     bp                      ;restore caller's stack frame
        ret
_WritePixelX    endp
        end
```

## LISTING 47.3 L47-3.ASM

```
; Mode X (320x240, 256 colors) read pixel routine. Works on all VGAs.
; No clipping is performed.
; C near-callable as:
;
;    unsigned int ReadPixelX(int X, int Y, unsigned int PageBase);

GC_INDEX        equ   03ceh             ;Graphics Controller Index
READ_MAP        equ   04h               ;index in GC of the Read Map register
SCREEN_SEG      equ   0a000h            ;segment of display memory in mode X
SCREEN_WIDTH    equ   80                ;width of screen in bytes from one scan line
                                        ; to the next
parms   struc
        dw      2 dup (?)               ;pushed BP and return address
X       dw      ?                       ;X coordinate of pixel to read
Y       dw      ?                       ;Y coordinate of pixel to read
PageBase dw     ?                       ;base offset in display memory of page from
                                        ; which to read pixel
parms   ends

        .model  small
        .code
        public  _ReadPixelX
_ReadPixelX     proc    near
        push    bp                      ;preserve caller's stack frame
        mov     bp,sp                   ;point to local stack frame
```

```
        mov     ax,SCREEN_WIDTH
        mul     [bp+Y]                  ;offset of pixel's scan line in page
        mov     bx,[bp+X]
        shr     bx,1
        shr     bx,1                    ;X/4 = offset of pixel in scan line
        add     bx,ax                   ;offset of pixel in page
        add     bx,[bp+PageBase]        ;offset of pixel in display memory
        mov     ax,SCREEN_SEG
        mov     es,ax                   ;point ES:BX to the pixel's address

        mov     ah,byte ptr [bp+X]
        and     ah,011b                 ;AH = pixel's plane
        mov     al,READ_MAP             ;AL = index in GC of the Read Map reg
        mov     dx,GC_INDEX             ;set the Read Map to read the pixel's
        out     dx,ax                   ; plane

        mov     al,es:[bx]              ;read the pixel's color
        sub     ah,ah                   ;convert it to an unsigned int

        pop     bp                      ;restore caller's stack frame
        ret
_ReadPixelX     endp
        end
```

## Designing from a Mode X Perspective

Listing 47.4 shows Mode X rectangle fill code. The plane is selected for each pixel in turn, with drawing cycling from plane 0 to plane 3, then wrapping back to plane 0. This is the sort of code that stems from a write-pixel line of thinking; it reflects not a whit of the unique perspective that Mode X demands, and although it looks reasonably efficient, it is in fact some of the slowest graphics code you will ever see. I've provided Listing 47.4 partly for illustrative purposes, but mostly so we'll have a point of reference for the substantial speed-up that's possible with code that's designed from a Mode X perspective.

### LISTING 47.4 L47-4.ASM

```
; Mode X (320x240, 256 colors) rectangle fill routine. Works on all
; VGAs. Uses slow approach that selects the plane explicitly for each
; pixel. Fills up to but not including the column at EndX and the row
; at EndY. No clipping is performed.
; C near-callable as:
;
;    void FillRectangleX(int StartX, int StartY, int EndX, int EndY,
;       unsigned int PageBase, int Color);

SC_INDEX        equ     03c4h           ;Sequence Controller Index
MAP_MASK        equ     02h             ;index in SC of Map Mask register
SCREEN_SEG      equ     0a000h          ;segment of display memory in mode X
SCREEN_WIDTH    equ     80              ;width of screen in bytes from one scan line
                                        ; to the next
parms   struc
        dw      2 dup (?)               ;pushed BP and return address
StartX  dw      ?               ;X coordinate of upper left corner of rect
StartY  dw      ?               ;Y coordinate of upper left corner of rect
EndX    dw      ?               ;X coordinate of lower right corner of rect
                                ; (the row at EndX is not filled)
```

```
EndY    dw      ?                       ;Y coordinate of lower right corner of rect
                                        ; (the column at EndY is not filled)
PageBase dw     ?                       ;base offset in display memory of page in
                                        ; which to fill rectangle
Color   dw      ?                       ;color in which to draw pixel
parms   ends

        .model  small
        .code
        public  _FillRectangleX
_FillRectangleX proc    near
        push    bp                      ;preserve caller's stack frame
        mov     bp,sp                   ;point to local stack frame
        push    si                      ;preserve caller's register variables
        push    di

        mov     ax,SCREEN_WIDTH
        mul     [bp+StartY]             ;offset in page of top rectangle scan line
        mov     di,[bp+StartX]
        shr     di,1
        shr     di,1                    ;X/4 = offset of first rectangle pixel in scan
                                        ; line
        add     di,ax                   ;offset of first rectangle pixel in page
        add     di,[bp+PageBase]        ;offset of first rectangle pixel in
                                        ; display memory
        mov     ax,SCREEN_SEG
        mov     es,ax                   ;point ES:DI to the first rectangle pixel's
                                        ; address
        mov     dx,SC_INDEX             ;set the Sequence Controller Index to
        mov     al,MAP_MASK             ; point to the Map Mask register
        out     dx,al
        inc     dx                      ;point DX to the SC Data register
        mov     cl,byte ptr [bp+StartX]
        and     cl,011b                 ;CL = first rectangle pixel's plane
        mov     al,01h
        shl     al,cl                   ;set only the bit for the pixel's plane to 1
        mov     ah,byte ptr [bp+Color]  ;color with which to fill
        mov     bx,[bp+EndY]
        sub     bx,[bp+StartY]          ;BX = height of rectangle
        jle     FillDone                ;skip if 0 or negative height
        mov     si,[bp+EndX]
        sub     si,[bp+StartX]          ;CX = width of rectangle
        jle     FillDone                ;skip if 0 or negative width
FillRowsLoop:
        push    ax                      ;remember the plane mask for the left edge
        push    di                      ;remember the start offset of the scan line
        mov     cx,si                   ;set count of pixels in this scan line
FillScanLineLoop:
        out     dx,al                   ;set the plane for this pixel
        mov     es:[di],ah              ;draw the pixel
        shl     al,1                    ;adjust the plane mask for the next pixel's
        and     al,01111b               ; bit, modulo 4
        jnz     AddressSet              ;advance address if we turned over from
        inc     di                      ; plane 3 to plane 0
        mov     al,00001b               ;set plane mask bit for plane 0
AddressSet:
        loop    FillScanLineLoop
        pop     di                      ;retrieve the start offset of the scan line
        add     di,SCREEN_WIDTH         ;point to the start of the next scan
                                        ; line of the rectangle
```

```
        pop     ax                          ;retrieve the plane mask for the left edge
        dec     bx                          ;count down scan lines
        jnz     FillRowsLoop
FillDone:
        pop     di                          ;restore caller's register variables
        pop     si
        pop     bp                          ;restore caller's stack frame
        ret
_FillRectangleX endp
        end
```

The two major weaknesses of Listing 47.4 both result from selecting the plane on a pixel by pixel basis. First, endless **OUT**s (which are particularly slow on 386s, 486s, and Pentiums, much slower than accesses to display memory) must be performed, and, second, **REP STOS** can't be used. Listing 47.5 overcomes both these problems by tailoring the fill technique to the organization of display memory. Each plane is filled in its entirety in one burst before the next plane is processed, so only five **OUT**s are required in all, and **REP STOS** can indeed be used; I've used **REP STOSB** in Listings 47.5 and 47.6. **REP STOSW** could be used and would improve performance on most VGAs; however, **REP STOSW** requires extra overhead to set up, so it can be slower for small rectangles, especially on 8-bit VGAs. Note that doing an entire plane at a time can produce a "fading-in" effect for large images, because all columns for one plane are drawn before any columns for the next. If this is a problem, the four planes can be cycled through once for each scan line, rather than once for the entire rectangle.

Listing 47.5 is 2.5 times faster than Listing 47.4 at clearing the screen on a 20-MHz cached 386 with a Paradise VGA. Although Listing 47.5 is slightly slower than an equivalent mode 13H fill routine would be, it's not grievously so.

*In general, performing plane-at-a-time operations can make almost any Mode X operation, at the worst, nearly as fast as the same operation in mode 13H (although this sort of Mode X programming is admittedly fairly complex). In this pursuit, it can help to organize data structures with Mode X in mind. For example, icons could be prearranged in system memory with the pixels organized into four plane-oriented sets (or, again, in four sets per scan line to avoid a fading-in effect) to facilitate copying to the screen a plane at a time with **REP MOVS**.*

### LISTING 47.5 L47-5.ASM

```
; Mode X (320x240, 256 colors) rectangle fill routine. Works on all
; VGAs. Uses medium-speed approach that selects each plane only once
; per rectangle; this results in a fade-in effect for large
; rectangles. Fills up to but not including the column at EndX and the
; row at EndY. No clipping is performed.
; C near-callable as:
;
;    void FillRectangleX(int StartX, int StartY, int EndX, int EndY,
;       unsigned int PageBase, int Color);

SC_INDEX        equ     03c4h           ;Sequence Controller Index
MAP_MASK        equ     02h             ;index in SC of Map Mask register
SCREEN_SEG      equ     0a000h          ;segment of display memory in mode X
```

```
SCREEN_WIDTH     equ   80               ;width of screen in bytes from one scan line
                                        ; to the next
parms struc
          dw      2 dup (?)             ;pushed BP and return address
StartX    dw      ?                     ;X coordinate of upper left corner of rect
StartY    dw      ?                     ;Y coordinate of upper left corner of rect
EndX      dw      ?                     ;X coordinate of lower right corner of rect
                                        ; (the row at EndX is not filled)
EndY      dw      ?                     ;Y coordinate of lower right corner of rect
                                        ; (the column at EndY is not filled)
PageBase  dw     ?                      ;base offset in display memory of page in
                                        ; which to fill rectangle
Color     dw      ?                     ;color in which to draw pixel
parms ends

StartOffset      equ   -2               ;local storage for start offset of rectangle
Width            equ   -4               ;local storage for address width of rectangle
Height           equ   -6               ;local storage for height of rectangle
PlaneInfo        equ   -8               ;local storage for plane # and plane mask
STACK_FRAME_SIZE equ   8

        .model  small
        .code
        public  _FillRectangleX
_FillRectangleX proc    near
        push    bp                      ;preserve caller's stack frame
        mov     bp,sp                   ;point to local stack frame
        sub     sp,STACK_FRAME_SIZE     ;allocate space for local vars
        push    si                      ;preserve caller's register variables
        push    di

        cld
        mov     ax,SCREEN_WIDTH
        mul     [bp+StartY]             ;offset in page of top rectangle scan line
        mov     di,[bp+StartX]
        shr     di,1
        shr     di,1                    ;X/4 = offset of first rectangle pixel in scan
                                        ; line
        add     di,ax                   ;offset of first rectangle pixel in page
        add     di,[bp+PageBase]        ;offset of first rectangle pixel in
                                        ; display memory
        mov     ax,SCREEN_SEG
        mov     es,ax                   ;point ES:DI to the first rectangle pixel's
        mov     [bp+StartOffset],di     ; address
        mov     dx,SC_INDEX             ;set the Sequence Controller Index to
        mov     al,MAP_MASK             ; point to the Map Mask register
        out     dx,al
        mov     bx,[bp+EndY]
        sub     bx,[bp+StartY]          ;BX = height of rectangle
        jle     FillDone                ;skip if 0 or negative height
        mov     [bp+Height],bx
        mov     dx,[bp+EndX]
        mov     cx,[bp+StartX]
        cmp     dx,cx
        jle     FillDone                ;skip if 0 or negative width
        dec     dx
        and     cx,not 011b
        sub     dx,cx
        shr     dx,1
        shr     dx,1
```

```
        inc     dx                      ;# of addresses across rectangle to fill
        mov     [bp+Width],dx
        mov     word ptr [bp+PlaneInfo],0001h
                                        ;lower byte = plane mask for plane 0,
                                        ; upper byte = plane # for plane 0
FillPlanesLoop:
        mov     ax,word ptr [bp+PlaneInfo]
        mov     dx,SC_INDEX+1           ;point DX to the SC Data register
        out     dx,al                   ;set the plane for this pixel
        mov     di,[bp+StartOffset]     ;point ES:DI to rectangle start
        mov     dx,[bp+Width]
        mov     cl,byte ptr [bp+StartX]
        and     cl,011b                 ;plane # of first pixel in initial byte
        cmp     ah,cl                   ;do we draw this plane in the initial byte?
        jae     InitAddrSet             ;yes
        dec     dx                      ;no, so skip the initial byte
        jz      FillLoopBottom          ;skip this plane if no pixels in it
        inc     di
InitAddrSet:
        mov     cl,byte ptr [bp+EndX]
        dec     cl
        and     cl,011b                 ;plane # of last pixel in final byte
        cmp     ah,cl                   ;do we draw this plane in the final byte?
        jbe     WidthSet                ;yes
        dec     dx                      ;no, so skip the final byte
        jz      FillLoopBottom          ;skip this planes if no pixels in it
WidthSet:
        mov     si,SCREEN_WIDTH
        sub     si,dx                   ;distance from end of one scan line to start
                                        ; of next
        mov     bx,[bp+Height]          ;# of lines to fill
        mov     al,byte ptr [bp+Color]  ;color with which to fill
FillRowsLoop:
        mov     cx,dx                   ;# of bytes across scan line
        rep     stosb                   ;fill the scan line in this plane
        add     di,si                   ;point to the start of the next scan
                                        ; line of the rectangle
        dec     bx                      ;count down scan lines
        jnz     FillRowsLoop
FillLoopBottom:
        mov     ax,word ptr [bp+PlaneInfo]
        shl     al,1                    ;set the plane bit to the next plane
        inc     ah                      ;increment the plane #
        mov     word ptr [bp+PlaneInfo],ax
        cmp     ah,4                    ;have we done all planes?
        jnz     FillPlanesLoop          ;continue if any more planes
FillDone:
        pop     di                      ;restore caller's register variables
        pop     si
        mov     sp,bp                   ;discard storage for local variables
        pop     bp                      ;restore caller's stack frame
        ret
_FillRectangleX endp
        end
```

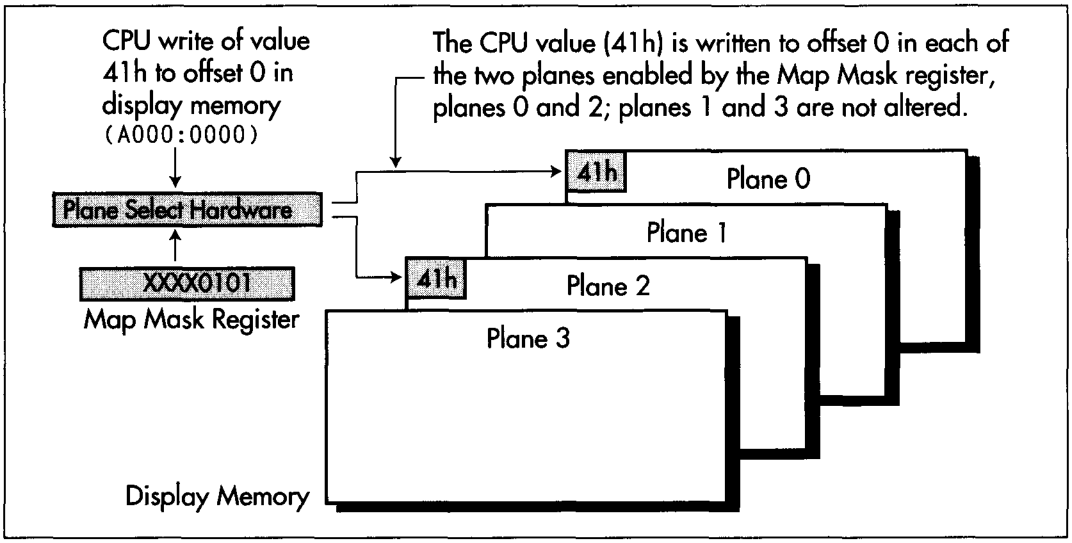
## Hardware Assist from an Unexpected Quarter

Listing 47.5 illustrates the benefits of designing code from a Mode X perspective; this is the software aspect of Mode X optimization, which suffices to make Mode X about as fast as mode 13H. That alone makes Mode X an attractive mode, given its square pixels, page flipping, and offscreen memory, but superior performance would nonetheless be a pleasant addition to that list. Superior performance is indeed possible in Mode X, although, oddly enough, it comes courtesy of the VGA's hardware, which was never designed to be used in 256-color modes.

All of the VGA's hardware assist features are available in Mode X, although some are not particularly useful. The VGA hardware feature that's truly the key to Mode X performance is the ability to process four planes' worth of data in parallel; this includes both the latches and the capability to fan data out to any or all planes. For rectangular fills, we'll just need to fan the data out to various planes, so I'll defer a discussion of other hardware features for now. (By the way, the ALUs, bit mask, and most other VGA hardware features are also available in mode 13H—but parallel data processing is not.)

In planar modes, such as Mode X, a byte written by the CPU to display memory may actually go to anywhere between zero and four planes, as shown in Figure 47.2. Each plane for which the setting of the corresponding bit in the Map Mask register is 1 receives the CPU data, and each plane for which the corresponding bit is 0 is not modified.

In 16-color modes, each plane contains one-quarter of each of eight pixels, with the 4 bits of each pixel spanning all four planes. Not so in Mode X. Look at Figure 47.1 again; each plane contains one pixel in its entirety, with four pixels at any given address, one per plane. Still, the Map Mask register does the same job in Mode X as




*Selecting planes with the Map Mask register.*
**Figure 47.2**

in 16-color modes; set it to 0FH (all 1-bits), and all four planes will be written to by each CPU access. Thus, it would seem that up to four pixels could be set by a single Mode X byte-sized write to display memory, potentially speeding up operations like rectangle fills by four times.

And, as it turns out, four-plane parallelism works quite nicely indeed. Listing 47.6 is yet another rectangle-fill routine, this time using the Map Mask to set up to four pixels per **STOS**. The only trick to Listing 47.6 is that any left or right edge that isn't aligned to a multiple-of-four pixel column (that is, a column at which one four-pixel set ends and the next begins) must be clipped via the Map Mask register, because not all pixels at the address containing the edge are modified. Performance is as expected; Listing 47.6 is nearly ten times faster at clearing the screen than Listing 47.4 and just about four times faster than Listing 47.5—and also about four times faster than the same rectangle fill in mode 13H. Understanding the bitmap organization and display hardware of Mode X does indeed pay.

Note that the return from Mode X's parallelism is not always 4×; some adapters lack the underlying memory bandwidth to write data that fast. However, Mode X parallel access should always be faster than mode 13H access; the only question on any given adapter is how *much* faster.

### LISTING 47.6 L47-6.ASM

```
; Mode X (320x240, 256 colors) rectangle fill routine. Works on all
; VGAs. Uses fast approach that fans data out to up to four planes at
; once to draw up to four pixels at once. Fills up to but not
; including the column at EndX and the row at EndY. No clipping is
; performed.
; C near-callable as:
;    void FillRectangleX(int StartX, int StartY, int EndX, int EndY,
;       unsigned int PageBase, int Color);

SC_INDEX        equ   03c4h           ;Sequence Controller Index
MAP_MASK        equ   02h             ;index in SC of Map Mask register
SCREEN_SEG      equ   0a000h          ;segment of display memory in mode X
SCREEN_WIDTH    equ   80              ;width of screen in bytes from one scan line
                                      ; to the next
parms   struc
        dw      2 dup (?)             ;pushed BP and return address
StartX  dw      ?                     ;X coordinate of upper left corner of rect
StartY  dw      ?                     ;Y coordinate of upper left corner of rect
EndX    dw      ?                     ;X coordinate of lower right corner of rect
                                      ; (the row at EndX is not filled)
EndY    dw      ?                     ;Y coordinate of lower right corner of rect
                                      ; (the column at EndY is not filled)
PageBase  dw    ?                     ;base offset in display memory of page in
                                      ; which to fill rectangle
Color   dw      ?                     ;color in which to draw pixel
parms   ends

        .model  small
        .data
; Plane masks for clipping left and right edges of rectangle.
LeftClipPlaneMask       db      00fh,00eh,00ch,008h
```

```
RightClipPlaneMask      db      00fh,001h,003h,007h
        .code
        public  _FillRectangleX
_FillRectangleX proc    near
        push    bp                      ;preserve caller's stack frame
        mov     bp,sp                   ;point to local stack frame
        push    si                      ;preserve caller's register variables
        push    di

        cld
        mov     ax,SCREEN_WIDTH
        mul     [bp+StartY]             ;offset in page of top rectangle scan line
        mov     di,[bp+StartX]
        shr     di,1                    ;X/4 = offset of first rectangle pixel in scan
        shr     di,1                    ; line
        add     di,ax                   ;offset of first rectangle pixel in page
        add     di,[bp+PageBase]        ;offset of first rectangle pixel in
                                        ; display memory
        mov     ax,SCREEN_SEG           ;point ES:DI to the first rectangle
        mov     es,ax                   ; pixel's address
        mov     dx,SC_INDEX             ;set the Sequence Controller Index to
        mov     al,MAP_MASK             ; point to the Map Mask register
        out     dx,al
        inc     dx                      ;point DX to the SC Data register
        mov     si,[bp+StartX]
        and     si,0003h                ;look up left edge plane mask
        mov     bh,LeftClipPlaneMask[si] ; to clip & put in BH
        mov     si,[bp+EndX]
        and     si,0003h                ;look up right edge plane
        mov     bl,RightClipPlaneMask[si] ; mask to clip & put in BL

        mov     cx,[bp+EndX]            ;calculate # of addresses across rect
        mov     si,[bp+StartX]
        cmp     cx,si
        jle     FillDone                ;skip if 0 or negative width
        dec     cx
        and     si,not 011b
        sub     cx,si
        shr     cx,1
        shr     cx,1                    ;# of addresses across rectangle to fill - 1
        jnz     MasksSet                ;there's more than one byte to draw
        and     bh,bl                   ;there's only one byte, so combine the left-
                                        ; and right-edge clip masks
MasksSet:
        mov     si,[bp+EndY]
        sub     si,[bp+StartY]          ;BX = height of rectangle
        jle     FillDone                ;skip if 0 or negative height
        mov     ah,byte ptr [bp+Color]  ;color with which to fill
        mov     bp,SCREEN_WIDTH         ;stack frame isn't needed any more
        sub     bp,cx                   ;distance from end of one scan line to start
        dec     bp                      ; of next
FillRowsLoop:
        push    cx                      ;remember width in addresses - 1
        mov     al,bh                   ;put left-edge clip mask in AL
        out     dx,al                   ;set the left-edge plane (clip) mask
        mov     al,ah                   ;put color in AL
        stosb                           ;draw the left edge
        dec     cx                      ;count off left edge byte
        js      FillLoopBottom          ;that's the only byte
        jz      DoRightEdge             ;there are only two bytes
```

```
        mov     al,00fh                 ;middle addresses are drawn 4 pixels at a pop
        out     dx,al                   ;set the middle pixel mask to no clip
        mov     al,ah                   ;put color in AL
        rep     stosb                   ;draw the middle addresses four pixels apiece
DoRightEdge:
        mov     al,bl                   ;put right-edge clip mask in AL
        out     dx,al                   ;set the right-edge plane (clip) mask
        mov     al,ah                   ;put color in AL
        stosb                           ;draw the right edge
FillLoopBottom:
        add     di,bp                   ;point to the start of the next scan line of
                                        ; the rectangle
        pop     cx                      ;retrieve width in addresses - 1
        dec     si                      ;count down scan lines
        jnz     FillRowsLoop
FillDone:
        pop     di                      ;restore caller's register variables
        pop     si
        pop     bp                      ;restore caller's stack frame
        ret
_FillRectangleX endp
        end
```

Just so you can see Mode X in action, Listing 47.7 is a sample program that selects Mode X and draws a number of rectangles. Listing 47.7 links to any of the rectangle fill routines I've presented.

And now, I hope, you're beginning to see why I'm so fond of Mode X. In the next chapter, we'll continue with Mode X by exploring the wonders that the latches and parallel plane hardware can work on scrolls, copies, blits, and pattern fills.

### LISTING 47.7 L47-7.C

```
/* Program to demonstrate mode X (320x240, 256-colors) rectangle
   fill by drawing adjacent 20x20 rectangles in successive colors from
   0 on up across and down the screen */
#include <conio.h>
#include <dos.h>

void Set320x240Mode(void);
void FillRectangleX(int, int, int, int, unsigned int, int);

void main() {
   int i,j;
   union REGS regset;

   Set320x240Mode();
   FillRectangleX(0,0,320,240,0,0); /* clear the screen to black */
   for (j = 1; j < 220; j += 21) {
      for (i = 1; i < 300; i += 21) {
         FillRectangleX(i, j, i+20, j+20, 0, ((j/21*15)+i/21) & 0xFF);
      }
   }
   getch();
   regset.x.ax = 0x0003;   /* switch back to text mode and done */
   int86(0x10, &regset, &regset);
}
```