Yet Another VGA Write Mode
Chapter 27

Write Mode 2, Chunky Bitmaps, and Text-Graphics Coexistence

In the last chapter, we learned about the markedly peculiar write mode 3 of the VGA, after having spent three chapters learning the ins and outs of the VGA's data path in write mode 0, touching on write mode 1 as well in Chapter 23. In all, the VGA supports four write modes—write modes 0, 1, 2, and 3—and read modes 0 and 1 as well. Which leaves two burning questions: What is write mode 2, and how the heck do you read VGA memory?

Write mode 2 is a bit unusual but not really hard to understand, particularly if you followed the description of set/reset in Chapter 25. Reading VGA memory, on the other hand, can be stranger than you could ever imagine.

Let's start with the easy stuff, write mode 2, and save the read modes for the next chapter.

Write Mode 2 and Set/Reset

Remember how set/reset works? Good, because that's pretty much how write mode 2 works. (You don't remember? Well, I'll provide a brief refresher, but I suggest that you go back through Chapters 23 through 25 and come up to speed on the VGA.)
Recall that the set/reset circuitry for each of the four planes affects the byte written by the CPU in one of three ways: By replacing the CPU byte with 0, by replacing it with 0FFH, or by leaving it unchanged. The nature of the transformation for each plane is controlled by two bits. The enable set/reset bit for a given plane selects whether the CPU byte is replaced or not, and the set/reset bit for that plane selects the value with which the CPU byte is replaced if the enable set/reset bit is 1. The net effect of set/reset is to independently force any, none, or all planes to either of all ones or all zeros on CPU writes. As we discussed in Chapter 25, this is a convenient way to force a specific color to appear no matter what color the pixels being overwritten are. Set/reset also allows the CPU to control the contents of some planes while the set/reset circuitry controls the contents of other planes.

Write mode 2 is basically a set/reset-type mode with enable set/reset always on for all planes and the set/reset data coming directly from the byte written by the CPU. Put another way, the lower four bits written by the CPU are written across the four planes, thereby becoming a color value. Put yet another way, bit 0 of the CPU byte is expanded to a byte and sent to the plane 0 ALU (if bit 0 is 0, a 0 byte is the CPU-side input to the plane 0 ALU, while if bit 0 is 1, a 0FFH byte is the CPU-side input); likewise, bit 1 of the CPU byte is expanded to a byte for plane 1, bit 2 is expanded for plane 2, and bit 3 is expanded for plane 3.

It’s possible that you understand write mode 2 thoroughly at this point; nonetheless, I suspect that some additional explanation of an admittedly non-obvious mode wouldn’t hurt. Let’s follow the CPU byte through the VGA in write mode 2, step by step.

A Byte’s Progress in Write Mode 2

Figure 27.1 shows the write mode 2 data path. The CPU byte comes into the VGA and is split into four separate bits, one for each plane. Bits 7-4 of the CPU byte vanish into the bit bucket, never to be heard from again. Speculation long held that those 4 unused bits indicated that IBM would someday come out with an 8-plane adapter that supported 256 colors. When IBM did finally come out with a 256-color mode (mode 13H of the VGA), it turned out not to be planar at all, and the upper nibble of the CPU byte remains unused in write mode 2 to this day.

The bit of the CPU byte sent to each plane is expanded to a 0 or 0FFH byte, depending on whether the bit is 0 or 1, respectively. The byte for each plane then becomes the CPU-side input to the respective plane’s ALU. From this point on, the write mode 2 data path is identical to the write mode 0 data path. As discussed in earlier articles, the latch byte for each plane is the other ALU input, and the ALU either ANDs, ORs, or XORs the two bytes together or simply passes the CPU-side byte through. The byte generated by each plane’s ALU then goes through the bit mask circuitry, which selects on a bit-by-bit basis between the ALU byte and the latch byte. Finally, the byte from the bit mask circuitry for each plane is written to that plane if the corresponding bit in the Map Mask register is set to 1.
It's worth noting two differences between write mode 2 and write mode 0, the standard write mode of the VGA. First, rotation of the CPU data byte does not take place in write mode 2. Second, the Set/Reset and Enable Set/Reset registers have no effect in write mode 2.

Now that we understand the mechanics of write mode 2, we can step back and get a feel for what it might be useful for. View bits 3-0 of the CPU byte as a single pixel in...
one of 16 colors. Next imagine that nibble turned sideways and written across the four planes, one bit to a plane. Finally, expand each of the bits to a byte, as shown in Figure 27.2, so that 8 pixels are drawn in the color selected by bits 3-0 of the CPU byte. Within the constraints of the VGA’s data paths, that’s exactly what write mode 2 does. By “the constraints of the VGA’s data paths,” I mean the ALUs, the bit mask, and the map mask. As Figure 27.1 indicates, the ALUs can modify the color written by the CPU, the map mask can prevent the CPU from altering selected planes, and the bit mask can prevent the CPU from altering selected bits of the byte written to. (Actually, the bit mask simply substitutes latch bits for ALU bits, but since the latches are normally loaded from the destination display memory byte, the net effect of the bit mask is usually to preserve bits of the destination byte.) These are not really constraints at all, of course, but rather features of the VGA; I simply want to make it clear that the use of write mode 2 to set 8 pixels to a given color is a rather simple special case among the many possible ways in which write mode 2 can be used to feed data into the VGA’s data path.

Write mode 2 is selected by setting bits 1 and 0 of the Graphics Mode register (Graphics Controller register 5) to 1 and 0, respectively. Since VGA registers are readable, the correct way to select write mode 2 on the VGA is to read the Graphics Mode register, mask off bits 1 and 0, OR in 00000010b (02H), and write the result back to the Graphics Mode register, thereby leaving the other bits in the register undisturbed.

Copying Chunky Bitmaps to VGA Memory Using Write Mode 2

Let’s take a look at two examples of write mode 2 in action. Listing 27.1 presents a program that uses write mode 2 to copy a graphics image in chunky format to the VGA. In chunky format adjacent bits in a single byte make up each pixel: mode 4 of the CGA, EGA, and VGA is a 2-bit-per-pixel chunky mode, and mode 13H of the VGA is an 8-bit-per-pixel chunky mode. Chunky format is convenient, since all the information about each pixel is contained in a single byte; consequently chunky format is often used to store bitmaps in system memory.

Unfortunately, VGA memory is organized as a planar rather than chunky bitmap in modes 0DH through 12H, with the bits that make up each pixel spread across four planes. The conversion from chunky to planar format in write mode 0 is quite a nuisance, requiring a good deal of bit manipulation. In write mode 2, however, the conversion becomes a snap, as shown in Listing 27.1. Once the VGA is placed in write mode 2, the lower four bits (the lower nibble) of the CPU byte (a single 4-bit chunky pixel) become eight planar pixels, all the same color. As discussed in Chapter 25, the bit mask makes it possible to narrow the effect of the CPU write down to a single pixel.

Given the above, conversion of a chunky 4-bit-per-pixel bitmap to the VGA’s planar format in write mode 2 is trivial. First, the Bit Mask register is set to allow only the VGA display memory bits corresponding to the leftmost chunky pixel of the two
stored in the first chunky bitmap byte to be modified. Next, the destination byte in
display memory is read in order to load the latches. Then a byte containing two
chunky pixels is read from the chunky bitmap in system memory, and the byte is
rotated four bits to the right to get the leftmost chunky pixel in position. This ro-
tated byte is written to the destination byte; since write mode 2 is active, each bit of
the chunky pixel goes to its respective plane, and since the Bit Mask register is set up
to allow only one bit in each plane to be modified, a single pixel in the color of the
chunky pixel is written to VGA memory.

This process is then repeated for the rightmost chunky pixel, if necessary, and re-
peated again for as many pixels as there are in the image.

**LISTING 27.1 L27-1.ASM**

: Program to illustrate one use of write mode 2 of the VGA and EGA by
: animating the image of an "A" drawn by copying it from a chunky
: bit-map in system memory to a planar bit-map in VGA or EGA memory.
:
: Assemble with MASM or TASM
:
: By Michael Abrash
:
Stack segment para stack 'STACK'
  db 512 dup(0)
Stack ends

SCREEN_WIDTH_IN_BYTES   equ 80
DISPLAY_MEMORY_SEGMENT   equ 0a000h
SC_INDEX                equ 3c4h  ;Sequence Controller Index register
MAP_MASK                equ 2     ;index of Map Mask register
GC_INDEX                equ 03ceh ;Graphics Controller Index reg
GRAPHICS_MODE           equ 5     ;index of Graphics Mode reg
BIT_MASK                equ 8     ;index of Bit Mask reg

Data segment para common 'DATA'
:
: Current location of "A" as it is animated across the screen.
:
CurrentX dw ?
CurrentY dw ?
RemainingLength dw ?
:
: Chunky bit-map image of a yellow "A" on a bright blue background
:
AImage label byte
  dw 13, 13 ;width, height in pixels
  db 000h, 000h, 000h, 000h, 000h, 000h, 000h
  db 009h, 099h, 099h, 099h, 099h, 099h, 099h
  db 009h, 099h, 099h, 099h, 099h, 099h, 099h
  db 009h, 099h, 099h, 099h, 099h, 099h, 099h
  db 009h, 099h, 099h, 099h, 099h, 099h, 099h
  db 009h, 099h, 099h, 099h, 099h, 099h, 099h
  db 009h, 099h, 099h, 099h, 099h, 099h, 099h
  db 009h, 099h, 099h, 099h, 099h, 099h, 099h
  db 009h, 099h, 099h, 099h, 099h, 099h, 099h
  db 009h, 099h, 099h, 099h, 099h, 099h, 099h
  db 009h, 099h, 099h, 099h, 099h, 099h, 099h
  db 009h, 099h, 099h, 099h, 099h, 099h, 099h

Yet Another VGA Write Mode  505
Data ends

Code segment para public 'CODE'
assume cs:Code, ds:Data
Start proc near
    mov ax, Data
    mov ds, ax
    mov ax, 10h
    int 10h ; select video mode 10h (640x350)

; Prepare for animation.

    mov [CurrentX], 0
    mov [CurrentY], 200
    mov [RemainingLength], 600 ; move 600 times

; Animate, repeating RemainingLength times. It's unnecessary to erase
; the old image, since the one pixel of blank fringe around the image
; erases the part of the old image not overlapped by the new image.

AnimationLoop:
    mov bx, [CurrentX]
    mov cx, [CurrentY]
    mov si, offset Aimage
    call DrawFromChunkyBitmap ; draw the "A" image
    inc [CurrentX] ; move one pixel to the right
    mov cx, 0 ; delay so we don't move the

DelayLoop: ; image too fast; adjust as
    loop DelayLoop ; needed

    dec [RemainingLength]
    jnz AnimationLoop ; Wait for a key before returning to text mode and ending.

    mov ah, 01h
    int 21h
    mov ax, 03h
    int 10h
    mov ah, 04ch
    int 21h
Start endp

; Draw an image stored in a chunky-bit map into planar VGA/EGA memory
; at the specified location.

; Input:
; BX - X screen location at which to draw the upper-left corner
; of the image
; CX - Y screen location at which to draw the upper-left corner
; of the image
; DS:SI - pointer to chunky image to draw, as follows:
; word at 0: width of image, in pixels
; word at 2: height of image, in pixels

506 Chapter 27
DrawFromChunkyBitmap proc near
cld

; Select write mode 2.

mov dx, GC_INDEX
mov al, GRAPHICS_MODE
out dx, al
inc dx
mov al, 02h
out dx, al

; Enable writes to all 4 planes.

mov dx, SC_INDEX
mov al, MAP_MASK
out dx, al
inc dx
mov al, 0fh
out dx, al

; Point ES:DI to the display memory byte in which the first pixel
; of the image goes, with AH set up as the bit mask to access that
; pixel within the addressed byte.

mov ax, SCREEN_WIDTH_IN_BYTES
mul cx  ; offset of start of top scan line
mov dx, ax
mov cl, bl
and cl, llb
mov ah, 80h  ; set AH to the bit mask for the original pixel
shr ah, cl ; initial pixel
shr bx, 1
shr bx, 1
shr bx, 1 ; x in bytes
add dx, bx
mov bx, DISPLAY_MEMORY_SEGMENT
mov es, bx  ; ES:DI points to the byte at which the
            ; upper left of the image goes

; Get the width and height of the image.

mov cx, [si] ; get the width
inc si
inc si
mov bx, [si] ; get the height
inc si
inc si
mov dx, GC_INDEX
mov al, BIT_MASK
out dx, al ; leave the GC Index register pointing
inc dx ; to the Bit Mask register
RowLoop:
  push ax ;preserve the left column's bit mask
  push cx ;preserve the width
  push di ;preserve the destination offset

ColumnLoop:
  mov al,ah
  out dx,al ;set the bit mask to draw this pixel
  mov al,es:[di] ;load the latches
  mov al,[si] ;get the next two chunky pixels
  shr al,1
  shr al,1
  shr al,1
  shr al,1 ;move the first pixel into the lsb
  stosb ;draw the first pixel
  ror ah,1 ;move mask to next pixel position
  jc CheckMorePixels ;is next pixel in the adjacent byte?
  dec di ;no

CheckMorePixels:
  dec cx ;see if there are any more pixels
  jz AdvanceToNextScanLine ; across in image
  mov al,ah
  out dx,al ;set the bit mask to draw this pixel
  mov al,es:[di] ;load the latches
  lodsb ;get the same two chunky pixels again
          ;and advance pointer to the next two pixels
  stosb ;draw the second of the two pixels
  ror ah,1 ;move mask to next pixel position
  jc CheckMorePixels2 ;is next pixel in the adjacent byte?
  dec di ;no

CheckMorePixels2:
  loop ColumnLoop ;see if there are any more pixels
                   ; across in the image
  jmp short CheckMoreScanLines

AdvanceToNextScanLine:
  inc si ;advance to the start of the next
          ; scan line in the image

CheckMoreScanLines:
  pop di ;get back the destination offset
  pop cx ;get back the width
  pop ax ;get back the left column's bit mask
  add di,SCREEN_WIDTH_IN BYTES
          ;point to the start of the next scan
          ; line of the image
  dec bx ;see if there are any more scan lines
  jnz RowLoop ; in the image
  ret

{DrawFromChunkyBitmap endp

Code ends
end Start

"That's an interesting application of write mode 2," you may well say, "but is it really useful?" While the ability to convert chunky bitmaps into VGA bitmaps does have its uses, Listing 27.1 is primarily intended to illustrate the mechanics of write mode 2.
For performance, it's best to store 16-color bitmaps in pre-separated four-plane format in system memory, and copy one plane at a time to the screen. Ideally, such bitmaps should be copied one scan line at a time, with all four planes completed for one scan line before moving on to the next. I say this because when entire images are copied one plane at a time, nasty transient color effects can occur as one plane becomes visibly changed before other planes have been modified.

Drawing Color-Patterned Lines Using Write Mode 2

A more serviceable use of write mode 2 is shown in the program presented in Listing 27.2. The program draws multicolored horizontal, vertical, and diagonal lines, basing the color patterns on passed color tables. Write mode 2 is ideal because in this application color can vary from one pixel to the next, and in write mode 2 all that's required to set pixel color is a change of the lower nibble of the byte written by the CPU. Set/reset could be used to achieve the same result, but an index/data pair of OUTs would be required to set the Set/Reset register to each new color. Similarly, the Map Mask register could be used in write mode 0 to set pixel color, but in this case not only would an index/data pair of OUTs be required but there would also be no guarantee that data already in display memory wouldn't interfere with the color of the pixel being drawn, since the Map Mask register allows only selected planes to be drawn to.

Listing 27.2 is hardly a comprehensive line drawing program. It draws only a few special line cases, and although it is reasonably fast, it is far from the fastest possible code to handle those cases, because it goes through a dot-plot routine and because it draws horizontal lines a pixel rather than a byte at a time. Write mode 2 would, however, serve just as well in a full-blown line drawing routine. For any type of patterned line drawing on the VGA, the basic approach remains the same: Use the bit mask to select the pixel (or pixels) to be altered and use the CPU byte in write mode 2 to select the color in which to draw.

**LISTING 27.2 L27-2.ASM**

: Program to illustrate one use of write mode 2 of the VGA and EGA by drawing lines in color patterns.

: Assemble with MASM or TASM

: By Michael Abrash

Stack segment para stack 'STACK'
    db 512 dup(0)
Stack ends

SCREEN_WIDTH_IN_BYTES equ 80
GRAPHICS_SEGMENT equ 0a000h ;mode 10 bit-map segment
SC_INDEX equ 3c4h ;Sequence Controller Index register
MAP_MASK equ 2 ;Index of Map Mask register
GC_INDEX equ 03ceh ;Graphics Controller Index reg
GRAPHICS_MODE equ 5 ;Index of Graphics Mode reg
BIT_MASK equ 8 ;Index of Bit Mask reg
Data segment para common 'DATA'
Pattern0 db 16
   db 0, 1, 2, 3, 4, 5, 6, 7, 8
   db 9, 10, 11, 12, 13, 14, 15
Pattern1 db 6
   db 2, 2, 2, 10, 10, 10
Pattern2 db 8
   db 15, 15, 15, 0, 0, 15, 0, 0
Pattern3 db 9
   db 1, 1, 2, 2, 2, 4, 4, 4
Data ends

Code segment para public 'CODE'
assume cs:Code, ds:Data
Start proc near
   mov ax, Data
   mov ds, ax
   mov ax, 10h
   int 10h ; select video mode 10h (640x350)
   : : Draw 8 radial lines in upper-left quadrant in pattern 0.
   : :
   mov bx, 0
   mov cx, 0
   mov si, offset Pattern0
   call QuadrantUp
   :
   : Draw 8 radial lines in upper-right quadrant in pattern 1.
   :
   mov bx, 320
   mov cx, 0
   mov si, offset Pattern1
   call QuadrantUp
   :
   : Draw 8 radial lines in lower-left quadrant in pattern 2.
   :
   mov bx, 0
   mov cx, 175
   mov si, offset Pattern2
   call QuadrantUp
   :
   : Draw 8 radial lines in lower-right quadrant in pattern 3.
   :
   mov bx, 320
   mov cx, 175
   mov si, offset Pattern3
   call QuadrantUp
   :
   : Wait for a key before returning to text mode and ending.
   :
   mov ah, 01h
   int 21h
   mov ax, 03h
   int 10h
   mov ah, 4ch
   int 21h
   :
   : Draws 8 radial lines with specified pattern in specified mode 10h quadrant.
; Input:
; BX = X coordinate of upper left corner of quadrant
; CX = Y coordinate of upper left corner of quadrant
; SI = pointer to pattern, in following form:
; Byte 0: Length of pattern
; Byte 1: Start of pattern, one color per byte
;
; AX, BX, CX, DX destroyed
;
; QuadrantUp proc near
; add bx,160
; add cx,87 ;point to the center of the quadrant
; mov ax,0
; mov dx,160
; mov ax,1
; mov dx,88
; mov ax,2
; mov dx,88
; mov ax,3
; mov dx,88
; mov ax,4
; mov dx,161
; mov ax,5
; mov dx,88
; mov ax,6
; mov dx,88
; mov ax,7
; mov dx,88
; ret
;
; QuadrantUp endp
;
; Draws a horizontal, vertical, or diagonal line (one of the eight possible radial lines) of the specified length from the specified starting point.
;
; Input:
; AX - line direction, as follows:
; 3 2 1
; 4 * 0
; 5 6 7
; BX = X coordinate of starting point
; CX = Y coordinate of starting point
; DX = length of line (number of pixels drawn)
;
; All registers preserved.
;
; Table of vectors to routines for each of the 8 possible lines.
;
; LineUpVectors label word
; dw LineUp0, LineUp1, LineUp2, LineUp3
; dw LineUp4, LineUp5, LineUp6, LineUp7
; Macro to draw horizontal, vertical, or diagonal line.
;
; Input:
; XParm = 1 to draw right, -1 to draw left, 0 to not move horz.
; YParm = 1 to draw up, -1 to draw down, 0 to not move vert.
; BX = X start location
; CX = Y start location
; DX = number of pixels to draw
; DS:SI = line pattern
;
MLLineUp macro XParm, YParm
    local   LineUpLoop, CheckMoreLine
    mov   di,si       ;set aside start offset of pattern
    lodsb            ;get length of pattern
    mov   ah,al
    LineUpLoop:
    lodsb            ;get color of this pixel...
    call   DotUpInColor   ;...and draw it
    if XParm EQ 1
        inc   bx
    endif
    if XParm EQ -1
        dec   bx
    endif
    if YParm EQ 1
        inc   cx
    endif
    if YParm EQ -1
        dec   cx
    endif
    dec   ah ;at end of pattern?
    jnz   CheckMoreLine   ;...and draw it
    mov   si,di            ;get back start of pattern
    mov   ah,al ;reset pattern count
CheckMoreLine:
    dec   dx
    jnz   LineUpLoop
    jmp   LineUpEnd
endm

LineUp proc near
    push ax
    push bx
    push cx
    push dx
    push si
    push di
    push es
    mov   di,ax
    mov   ax,GRAPhICS_SEGMENT
    mov   es,ax
    push dx           ;save line length

512 Chapter 27
; Enable writes to all planes.
;   mov   dx,SC_INDEX
mov   ax,MAP_MASK
out   dx,al
inc   dx
mov   ax,0fh
out   dx,al

; Select write mode 2.
;   mov   dx,GC_INDEX
mov   ax,GRAPHICS_MODE
out   dx,al
inc   dx
mov   al,02h
out   dx,al

; Vector to proper routine.
;   pop   dx          ;get back line length
shl   di,1
jmp   cs:[LineUpVectors+di]

; Horizontal line to right.
; LineUp0:
   MLineUp 1, 0
;
; Diagonal line to upper right.
; LineUp1:
   MLineUp 1, -1
;
; Vertical line to top.
; LineUp2:
   MLineUp 0, -1
;
; Diagonal line to upper left.
; LineUp3:
   MLineUp -1, -1
;
; Horizontal line to left.
; LineUp4:
   MLineUp -1, 0
;
; Diagonal line to bottom left.
; LineUp5:
   MLineUp -1, 1
;
; Vertical line to bottom.
; LineUp6:
   MLineUp 0, 1
LineUpEnd:
  pop  es
  pop  di
  pop  si
  pop  dx
  pop  cx
  pop  bx
  pop  ax
  ret
LineUp endp

.; Draws a dot in the specified color at the specified location.
.; Assumes that the VGA is in write mode 2 with writes to all planes
.; enabled and that ES points to display memory.
.; Input:
.;  AL = dot color
.;  BX = X coordinate of dot
.;  CX = Y coordinate of dot
.;  ES = display memory segment
.; All registers preserved.
.
DotUpInColor proc near
  push  bx
  push  cx
  push  dx
  push  di

  ; Point ES:DI to the display memory byte in which the pixel goes, with
  ; the bit mask set up to access that pixel within the addressed byte.

  push  ax             ;preserve dot color
  mov  ax,SCREEN_WIDTH_IN_BYTES
  mul  cx              ;offset of start of top scan line
  mov  di,ax
  mov  cl,bl
  and  cl,1111b
  mov  dx,6C_INDEX
  mov  al,BIT_MASK
  out  dx,al
  inc  dx
  mov  al,80h
  shr  al,cl
  out  dx,al          ;set the bit mask for the pixel
  shr  bx,1
  shr  bx,1
  shr  bx,1          ;X in bytes
  add  di,bx          ;offset of byte pixel is in
  mov  al,es:[di]    ;load latches
  pop  ax             ;get back dot color
  stosb              ;write dot in desired color
  pop  di
  pop  dx

DotUpInColor endp
When to Use Write Mode 2 and When to Use Set/Reset

As indicated earlier, write mode 2 and set/reset are functionally interchangeable. Write mode 2 lends itself to more efficient implementations when the drawing color changes frequently, as in Listing 27.2.

Set/reset tends to be superior when many pixels in succession are drawn in the same color, since with set/reset enabled for all planes the Set/Reset register provides the color data and as a result the CPU is free to draw whatever byte value it wishes. For example, the CPU can execute an OR instruction to display memory when set/reset is enabled for all planes, thus both loading the latches and writing the color value with a single instruction, secure in the knowledge that the value it writes is ignored in favor of the set/reset color.

Set/reset is also the mode of choice whenever it is necessary to force the value written to some planes to a fixed value while allowing the CPU byte to modify other planes. This is the mode of operation when set/reset is enabled for some but not all planes.

Mode 13H—320×200 with 256 Colors

I’m going to take a minute—and I do mean a minute—to discuss the programming model for mode 13H, the VGA’s 320×200 256-color mode. Frankly, there’s just not much to it, especially compared to the convoluted 16-color model that we’ve explored over the last five chapters. Mode 13H offers the simplest programming model in the history of PC graphics: A linear bitmap starting at A000:0000, consisting of 64,000 bytes, each controlling one pixel. The byte at offset 0 controls the upper left pixel on the screen, the byte at offset 319 controls the upper right pixel on the screen, the byte at offset 320 controls the second pixel down at the left of the screen, and the byte at offset 63,999 controls the lower right pixel on the screen. That’s all there is to it; it’s so simple that I’m not going to spend any time on a demo program, especially given that some of the listings later in this book, such as the antialiasing code in Chapter F on the companion CD-ROM, use mode 13H.

Flipping Pages from Text to Graphics and Back

A while back, I got an interesting letter from Phil Coleman, of La Jolla, who wrote: “Suppose I have the EGA in mode 10H (640×350 16-color graphics). I would like to
preserve some or all of the image while I temporarily switch to text mode 3 to give my user a 'Help' screen. Naturally memory is scarce so I'd rather not make a copy of the video buffer at A000H to 'remember' the image while I digress to the Help text. The EGA BIOS says that the screen memory will not be cleared on a mode set if bit 7 of AL is set. Yet if I try that, it is clear that writing text into the B800H buffer trashes much more than the 4K bytes of a text page; when I switch back to mode 10H, "ghosts" appear in the form of bands of colored dots. (When in text mode, I do make a copy of the 4K buffer at B800H before showing the help; and I restore the 4K before switching back to mode 10H.) Is there a way to preserve the graphics image while I switch to text mode?"

"A corollary to this question is: Where does the 64/128/256K of EGA memory 'hide' when the EGA is in text mode? Some I guess is used to store character sets, but what happens to the rest? Or rather, how can I protect it?"

Those are good questions. Alas, answering them in full would require extensive explanation that would have little general application, so I'm not going to do that. However, the issue of how to go to text mode and back without losing the graphics image certainly rates a short discussion, complete with some working code. That's especially true given that both the discussion and the code apply just as well to the VGA as to the EGA (with a few differences in mode 12H, the VGA's high-resolution mode, as noted below).

Phil is indeed correct in his observation that setting bit 7 of AL instructs the BIOS not to clear display memory on mode sets, and he is also correct in surmising that a font is loaded when going to text mode. The normal mode 10H bitmap occupies the first 28,000 bytes of each of the VGA's four planes. (The mode 12H bitmap takes up the first 38,400 bytes of each plane.) The normal mode 3 character/attribute memory map resides in the first 4000 bytes of planes 0 and 1 (the blue and green planes in mode 10H). The standard font in mode 3 is stored in the first 8K of plane 2 (the red plane in mode 10H). Neither mode 3 nor any other text mode makes use of plane 3 (the intensity plane in mode 10H); if necessary, plane 3 could be used as scratch memory in text mode.

Consequently, you can get away with saving a total of just under 16K bytes—the first 4000 bytes of planes 0 and 1 and the first 8K bytes of plane 2—when going from mode 10H or mode 12H to mode 3, to be restored on returning to graphics mode. That's hardly all there is to the matter of going from text to graphics and back without bitmap corruption, though. One interesting point is that the mode 10H bitmap can be relocated to A000:8000 simply by doing a mode set to mode 10H and setting the start address (programmed at CRT Controller registers 0CH and 0DH) to 8000H. You can then access display memory starting at A800:8000 instead of the normal A000:0000, with the resultant display exactly like that of normal mode 10H. There are BIOS issues, since the BIOS doesn't automatically access display memory at the
new start address, but if your program does all its drawing directly without the help of the BIOS, that’s no problem.

The mode 12H bitmap can’t start at A000:8000, because it’s so long that it would run off the end of display memory. However, the mode 12H bitmap can be relocated to, say, A000:6000, where it would fit without conflicting with the default font or the normal text mode memory map, although it would overlap two of the upper pages available for use (but rarely used) by text-mode programs.

At any rate, once the graphics mode bitmap is relocated, flipping to text mode and back becomes painless. The memory used by mode 3 doesn’t overlap the relocated mode 10H bitmap at all (unless additional portions of font memory are loaded), so all you need do is set bit 7 of AL on mode sets in order to flip back and forth between the two modes.

Another interesting point about flipping from graphics to text and back is that the standard mode 3 character/attribute map doesn’t actually take up every byte of the first 4000 bytes of planes 0 and 1. The standard mode 3 character/attribute map actually only takes up every even byte of the first 4000 in each plane; the odd bytes are left untouched. This means that only about 12K bytes actually have to be saved when going to text mode. The code in Listing 27.3 flips from graphics mode to text mode and back, saving only those 12K bytes that actually have to be saved. This code saves and restores the first 8K of plane 2 (the font area) while in graphics mode, but performs the save and restore of the 4000 bytes used for the character/attribute map while in text mode, because the characters and attributes, which are actually stored in the even bytes of planes 0 and 1, respectively, appear to be contiguous bytes in memory in text mode and so are easily saved as a single block.

Explaining why only every other byte of planes 0 and 1 is used in text mode and why characters and attributes appear to be contiguous bytes when they are actually in different planes is a large part of the explanation I’m not going to go into now. One bit of fallout from this, however, is that if you flip to text mode and preserve the graphics bitmap using the mechanism illustrated in Listing 27.3, you shouldn’t write to any text page other than page 0 (that is, don’t write to any offset in display memory above 3999 in text mode) or alter the Page Select bit in the Miscellaneous Output register (3C2H) while in text mode. In order to allow completely unfettered access to text pages, it would be necessary to save every byte in the first 32K of each of planes 0 and 1. (On the other hand, this would allow up to 16 text screens to be stored simultaneously, with any one displayable instantly.) Moreover, if any fonts other than the default font are loaded, the portions of plane 2 that those particular fonts are loaded into would have to be saved, up to a maximum of all 64K of plane 2. In the worst case, a full 128K would have to be saved in order to preserve all the memory potentially used by text mode.

As I said, Phil Coleman’s question is an interesting one, and I’ve only touched on the intriguing possibilities arising from the various configurations of display memory in
VGA graphics and text modes. Right now, though, we’ve still got the basics of the remarkably complex (but rewarding!) VGA to cover.

LISTING 27.3 L27-3.ASM

; Program to illustrate flipping from bit-mapped graphics mode to
; text mode and back without losing any of the graphics bit-map.
;
; Assemble with MASM or TASM
;
; By Michael Abrash
;
Stack segment para stack 'STACK'
    db  512 dup(0)
Stack ends

GRAPHICS-SEGMENT
TEXT_SEGMENT equ Oa000h ;mode 10 bit-map segment
TEXT_SEGMENT equ Ob800h ;mode 3 bit-map segment
SC_INDEX  equ 3c4h ;Sequence Controller Index register
MAP_MASK   equ 2 ;index of Map Mask register
GC_INDEX   equ 3ceh ;Graphics Controller Index register
READ_MAP   equ 4 ;index of Read Map register

Data segment para common 'DATA'

GStrikeAnyKeyMsg0 label byte
db  Odh, Oah, 'Graphics mode', Odh, Oah
    db 'Strike any key to continue...', Odh, Oah, '$'

GStrikeAnyKeyMsg1 label byte
db  Odh, Oah, 'Graphics mode again', Odh, Oah
    db 'Strike any key to continue...', Odh, Oah, '$'

TStrikeAnyKeyMsg label byte
    db  Odh, Oah, 'Text mode', Odh, Oah
    db 'Strike any key to continue...', Odh, Oah, '$'

Plane2Save db  2000h dup (?) ;save area for plane 2 data
            ;where font gets loaded
CharAttSave db  4000 dup (?) ;save area for memory wiped
            ;out by character/attribute
            ;data in text mode

Data ends

Code segment para public 'CODE'
assume cs:Code, ds:Data

Start proc near
    mov ax,10h
    int 10h            ;select video mode 10h (640x350)
    ; Fill the graphics bit-map with a colored pattern.
    ;
    cld
    mov  ax,GRAPHICS_SEGMENT
    mov  es,ax
    mov  ah,3            ;initial fill pattern
    mov  cx,4            ;four planes to fill
    mov  dx,SC_INDEX
    mov  al,MAP_MASK
    out  dx,al            ;leave the SC Index pointing to the
    inc  dx                ;Map Mask register
FillBitMap:
    mov al,10h
    shr al,cl
    out dx,al
    sub di,di
    mov al,ah
    push cx
    mov cx,8000h
    rep stosw
    pop cx
    shl ah,1
    mov a1,ah
    mov a1,ah
loop FillBitMap

; Put up "strike any key" message.

mov ax,Data
mov ds,ax
mov dx.offset GStrikeAnyKeyMsg
mov ah,9
int 21h

; Wait for a key.

mov ah,01h
int 21h

; Save the 8K of plane 2 that will be used by the font.

mov dx,GX_INDEX
mov al,READ_MAP
out dx,al
inc dx
mov al,2
out dx,al
mov ax,Data
mov es,ax
mov ax,GRAPHICS_SEGMENT
mov ds,ax
sub si,si
mov dx.offset PlaneZSave
mov cx,Z000h/2
rep movsw

; Go to text mode without clearing display memory.

mov ax,083h
int 10h

; Save the text mode bit-map.

mov ax,Data
mov es,ax
mov ax,TEXT_SEGMENT
mov ds,ax
sub si,si
mov dx.offset CharAttSave
mov cx,40000/2
rep movsw

Yet Another VGA Write Mode 519
; Fill the text mode screen with dots and put up "strike any key"
; message.

; mov ax,TEXT_SEGMENT
; mov es,ax
; sub di,di
; mov al,\.'.' ;fill character
; mov ah,7 ;fill attribute
; mov cx,4000/2 ;length of one text screen in words
; rep stosw
; mov ax,Data
; mov ds,ax
; mov dx,offset TStrikeAnyKeyMsg
; mov ah,9
; int 21h

; Wait for a key.

; mov ah,01h
; int 21h

; Restore the text mode screen to the state it was in on entering
; text mode.

; mov ax,Data
; mov ds,ax
; mov ax,TEXT_SEGMENT
; mov es,ax
; mov si,offset CharAttSave
; sub di,di
; mov cx,4000/2 ;length of one text screen in words
; rep movsw

; Return to mode 10h without clearing display memory.

; mov ax,90h
; int 10h

; Restore the portion of plane 2 that was wiped out by the font.

; mov dx,SC_INDEX
; mov al,MAP_MASK
; out dx,al
; inc dx
; mov al,4
; out dx,al ;set up to write to plane 2
; mov ax,Data
; mov ds,ax
; mov ax,GRAPHICS_SEGMENT
; mov es,ax
; mov si,offset Plane2Save
; sub di,di
; mov cx,2000h/2 ;restore 8K (length of default font)
; rep movsw

; Put up "strike any key" message.

; mov ax,Data
; mov ds,ax

520 Chapter 27
mov dx,offset GStrikeAnyKeyMsg
mov ah,9
int 21h

; Wait for a key before returning to text mode and ending.
;
mov ah,01h
int 21h
mov ax,03h
int 10h
mov ah,4ch
int 21h

Start endp

Code ends

end Start