Chapter 28

Reading VGA Memory

Chapter 28

Read Modes 0 and 1, and the Color Don't Care Register

Well, it's taken five chapters, but we've finally covered the data write path and all four write modes of the VGA. Now it's time to tackle the VGA's two read modes. While the read modes aren't as complex as the write modes, they're nothing to sneeze at. In particular, read mode 1 (also known as color compare mode) is rather unusual and not at all intuitive.

You may well ask, isn't *anything* about programming the VGA straightforward? Well...no. But then, clearing up the mysteries of VGA programming is what this part of the book is all about, so let's get started.

Read Mode 0

Read mode 0 is actually relatively uncomplicated, given that you understand the four-plane nature of the VGA. (If you don't understand the four-plane nature of the VGA, I strongly urge you to read Chapters 23–27 before continuing with this chapter.) Read mode 0, the read mode counterpart of write mode 0, lets you read from *one* (and only one) plane of VGA memory at any one time.

Read mode 0 is selected by setting bit 3 of the Graphics Mode register (Graphics Controller register 5) to 0. When read mode 0 is active, the plane that supplies the data when the CPU reads VGA memory is the plane selected by bits 1 and 0 of the

Read Map register (Graphics Controller register 4). When the Read Map register is set to 0, CPU reads come from plane 0 (the plane that normally contains blue pixel data). When the Read Map register is set to 1, CPU reads come from plane 1; when the Read Map register is 2, CPU reads come from plane 2; and when the Read Map register is 3, CPU reads come from plane 3.

That all seems simple enough; in read mode 0, the Read Map register acts as a selector among the four planes, determining which one of the planes will supply the value returned to the CPU. There is a slight complication, however, in that the value written to the Read Map register in order to read from a given plane is not the same as the value written to the Map Mask register (Sequence Controller register 2) in order to write to that plane.

Why is that? Well, in read mode 0, one and only one plane can be read at a time, so there are only four possible settings of the Read Map register: 0, 1, 2, or 3, to select reads from plane 0, 1, 2, or 3. In write mode 0, by contrast (in fact, in any write mode), any or all planes may be written to at once, since the byte written by the CPU can "fan out" to multiple planes. Consequently, there are not four but sixteen possible settings of the Map Mask register. The setting of the Map Mask register to write only to plane 0 is 1; to write only to plane 1 is 2; to write only to plane 2 is 4; and to write only to plane 3 is 8.

As you can see, the settings of the Read Map and Map Mask registers for accessing a given plane don't match. The code in Listing 28.1 illustrates this. Listing 28.1 simply copies a sixteen-color image from system memory to VGA memory, one plane at a time, then animates by repeatedly copying the image back to system memory, again one plane at a time, clearing the old image, and copying the image to a new location in VGA memory. Note the differing settings of the Read Map and Map Mask registers.

LISTING 28.1 L28-1.ASM

```
; Program to illustrate the use of the Read Map register in read mode 0.
; Animates by copying a 16-color image from VGA memory to system memory,
; one plane at a time, then copying the image back to a new location
; in VGA memory.
;
; By Michael Abrash
;
stack segment     word stack 'STACK'
      db    512 dup (?)
stack ends
;
data  segment     word 'DATA'
IMAGE_WIDTH EQU   4           ;in bytes
IMAGE_HEIGHT      EQU   32          ;in pixels
LEFT_BOUND  EQU   10          ;in bytes
RIGHT_BOUND EQU   66          ;in bytes
VGA_SEGMENT EQU   0a000h
SCREEN_WIDTH      EQU   80          ;in bytes
SC_INDEX    EQU   3c4h        ;Sequence Controller Index register
GC_INDEX    EQU   3ceh        ;Graphics Controller Index register
```

```
MAP_MASK    EQU   2           ;Map Mask register index in SC
READ_MAP    EQU   4           ;Read Map register index in GC
;
; Base pattern for 16-color image.
;
PatternPlane0     label byte
      db    32 dup (0ffh,0ffh,0,0)
PatternPlane1     label byte
      db    32 dup (0ffh,0,0ffh,0)
PatternPlane2     label byte
      db    32 dup (0f0h,0f0h,0f0h,0f0h)
PatternPlane3     label byte
      db    32 dup (0cch,0cch,0cch,0cch)
;
; Temporary storage for 16-color image during animation.
;
ImagePlane0 db    32*4 dup (?)
ImagePlane1 db    32*4 dup (?)
ImagePlane2 db    32*4 dup (?)
ImagePlane3 db    32*4 dup (?)
;
; Current image location & direction.
;
ImageX            dw    40    ;in bytes
ImageY            dw    100   ;in pixels
ImageXDirection   dw    1     ;in bytes
data  ends
;
code  segment     word 'CODE'
      assume      cs:code,ds:data
Start proc  near
      cld
      mov   ax,data
      mov   ds,ax
;
; Select graphics mode 10h.
;
      mov   ax,10h
      int   10h
;
; Draw the initial image.
;
      mov   si,offset PatternPlane0
      call  DrawImage
;
; Loop to animate by copying the image from VGA memory to system memory,
; erasing the image, and copying the image from system memory to a new
; location in VGA memory. Ends when a key is hit.
;
AnimateLoop:
;
; Copy the image from VGA memory to system memory.
;
      mov   di,offset ImagePlane0
      call  GetImage
;
; Clear the image from VGA memory.
;
      call  EraseImage
```

```
;
; Advance the image X coordinate, reversing direction if either edge
; of the screen has been reached.
;
      mov   ax,[ImageX]
      cmp   ax,LEFT_BOUND
      jz    ReverseDirection
      cmp   ax,RIGHT_BOUND
      jnz   SetNewX
ReverseDirection:
      neg   [ImageXDirection]
SetNewX:
      add   ax,[ImageXDirection]
      mov   [ImageX],ax
;
; Draw the image by copying it from system memory to VGA memory.
;
      mov   si,offset ImagePlane0
      call  DrawImage
;
; Slow things down a bit for visibility (adjust as needed).
;
      mov   cx,0
DelayLoop:
      loop  DelayLoop
;
; See if a key has been hit, ending the program.
;
      mov   ah,1
      int   16h
      jz          AnimateLoop
;
; Clear the key, return to text mode, and return to DOS.
;
      sub   ah,ah
      int   16h
      mov   ax,3
      int   10h
      mov   ah,4ch
      int   21h
Start endp
;
; Draws the image at offset DS:SI to the current image location in
; VGA memory.
;
DrawImage   proc  near
      mov   ax,VGA_SEGMENT
      mov   es,ax
      call  GetImageOffset    ;ES:DI is the destination address for the
                        ; image in VGA memory
      mov   dx,SC_INDEX
      mov   al,1        ;do plane 0 first
DrawImagePlaneLoop:
      push  di          ;image is drawn at the same offset in
                        ; each plane
      push  ax          ;preserve plane select
      mov   al,MAP_MASK ;Map Mask index
      out   dx,al       ;point SC Index to the Map Mask register
      pop   ax          ;get back plane select
      inc   dx          ;point to SC index register
```

```
      out   dx,al       ;set up the Map Mask to allow writes to
                        ; the plane of interest
      dec   dx          ;point back to SC Data register
      mov   bx,IMAGE_HEIGHT  ;# of scan lines in image
DrawImageLoop:
      mov   cx,IMAGE_WIDTH   ;# of bytes across image
      rep   movsb
      add   di,SCREEN_WIDTH-IMAGE_WIDTH
                        ;point to next scan line of image
      dec   bx          ;any more scan lines?
      jnz   DrawImageLoop
      pop   di          ;get back image start offset in VGA memory
      shl   al,1        ;Map Mask setting for next plane
      cmp   al,10h      ;have we done all four planes?
      jnz   DrawImagePlaneLoop
      ret
DrawImage   endp
;
; Copies the image from its current location in VGA memory into the
; buffer at DS:DI.
;
GetImage    proc  near
      mov   si,di       ;move destination offset into SI
      call  GetImageOffset   ;DI is offset of image in VGA memory
      xchg  si,di ;SI is offset of image, DI is destination offset
      push  ds
      pop   es          ;ES:DI is destination
      mov   ax,VGA_SEGMENT
      mov   ds,ax       ;DS:SI is source
;
      mov   dx,GC_INDEX
      sub   al,al       ;do plane 0 first
GetImagePlaneLoop:
      push  si          ;image comes from same offset in each plane
      push  ax          ;preserve plane select
      mov   al,READ_MAP ;Read Map index
      out   dx,al       ;point GC Index to Read Map register
      pop   ax          ;get back plane select
      inc   dx          ;point to GC Index register
      out   dx,al       ;set up the Read Map to select reads from
                        ; the plane of interest
      dec   dx          ;point back to GC data register
      mov   bx,IMAGE_HEIGHT  ;# of scan lines in image
GetImageLoop:
      mov   cx,IMAGE_WIDTH   ;# of bytes across image
      rep   movsb
      add   si,SCREEN_WIDTH-IMAGE_WIDTH
                        ;point to next scan line of image
      dec   bx          ;any more scan lines?
      jnz   GetImageLoop
      pop   si          ;get back image start offset
      inc   al          ;Read Map setting for next plane
      cmp   al,4        ;have we done all four planes?
      jnz   GetImagePlaneLoop
      push  es
      pop   ds          ;restore original DS
      ret
GetImage    endp
;
; Erases the image at its current location.
;
```

```
EraseImage proc  near
      mov   dx,SC_INDEX
      mov   al,MAP_MASK
      out   dx,al       ;point SC Index to the Map Mask register
      inc   dx          ;point to SC Data register
      mov   al,0fh
      out   dx,al       ;set up the Map Mask to allow writes to go to
                        ; all 4 planes
      mov   ax,VGA_SEGMENT
      mov   es,ax
      call  GetImageOffset    ;ES:DI points to the start address
                              ; of the image
      sub   al,al             ;erase with zeros
      mov   bx,IMAGE_HEIGHT   ;# of scan lines in image
EraseImageLoop:
      mov   cx,IMAGE_WIDTH    ;# of bytes across image
      rep   stosb
      add   di,SCREEN_WIDTH-IMAGE_WIDTH
                        ;point to next scan line of image
      dec   bx          ;any more scan lines?
      jnz   EraseImageLoop
      ret
EraseImage endp
;
; Returns the current offset of the image in the VGA segment in DI.
;
GetImageOffset   proc  near
      mov   ax,SCREEN_WIDTH
      mul   [ImageY]
      add   ax,[ImageX]
      mov   di,ax
      ret
GetImageOffset   endp
code  ends
      end   Start
```

By the way, the code in Listing 28.1 is intended only to illustrate read mode 0, and is, in general, a poor way to perform animation, since it's slow and tends to flicker. Later in this book, we'll take a look at some far better VGA animation techniques.

As you'd expect, neither the read mode nor the setting of the Read Map register affects CPU *writes* to VGA memory in any way.

An important point regarding reading VGA memory involves the VGA's latches. (Remember that each of the four latches stores a byte for one plane; on CPU writes, the latches can provide some or all of the data written to display memory, allowing fast copying and efficient pixel masking.) Whenever the CPU reads a given address in VGA memory, each of the four latches is loaded with the contents of the byte at that address in its respective plane. Even though the CPU only receives data from one plane in read mode 0, all four planes are always read, and the values read are stored in the latches. This is true in read mode 1 as well. In short, whenever the CPU reads VGA memory in any read mode, all four planes are read and all four latches are always loaded.

Read Mode 1

Read mode 0 is the workhorse read mode, but it's got an annoying limitation: Whenever you want to determine the color of a given pixel in read mode 0, you have to perform four VGA memory reads, one for each plane, and then interpret the four bytes you've read as eight 16-color pixels. That's a lot of programming. The code is also likely to run slowly, all the more so because a standard IBM VGA takes an average of 1.1 microseconds to complete each memory read, and read mode 0 requires four reads in order to read the four planes, not to mention the even greater amount of time taken by the **OUT**s required to switch between the planes. (1.1 microseconds may not sound like much, but on a 66-MHz 486, it's 73 clock cycles! Local-bus VGAs can be a good deal faster, but a read from the fastest local-bus adapter I've yet seen would still cost in the neighborhood of 10 486/66 cycles.)

Read mode 1, also known as *color compare mode*, provides special hardware assistance for determining whether a pixel is a given color. With a single read mode 1 read, you can determine whether each of up to eight pixels is a specific color, and you can even specify any or all planes as "don't care" planes in the pixel color comparison.

Read mode 1 is selected by setting bit 3 of the Graphics Mode register (Graphics Controller register 5) to 1. In its simplest form, read mode 1 compares the cross-plane value of each of the eight pixels at a given address to the color value in bits 3-0 of the Color Compare register (Graphics Controller register 2), and returns a 1 to the CPU in the bit position of each pixel that matches the color in the Color Compare register and a 0 for each pixel that does not match.

That's certainly interesting, but what's read mode 1 good for? One obvious application is in implementing flood-fill algorithms, since read mode 1 makes it easy to tell when a given byte contains a pixel of a boundary color. Another application is in detecting on-screen object collisions, as illustrated by the code in Listing 28.2.

LISTING 28.2 L28-2.ASM

```
; Program to illustrate use of read mode 1 (color compare mode)
; to detect collisions in display memory. Draws a yellow line on a
; blue background, then draws a perpendicular green line until the
; yellow line is reached.
;
; By Michael Abrash
;
stack segment     word stack 'STACK'
      db    512 dup (?)
stack ends
;
VGA_SEGMENT       EQU   0a000h
SCREEN_WIDTH            EQU   80    ;in bytes
GC_INDEX          EQU   3ceh        ;Graphics Controller Index register
SET_RESET         EQU   0           ;Set/Reset register index in GC
ENABLE_SET_RESET  EQU   1           ;Enable Set/Reset register index in GC
COLOR_COMPARE           EQU   2     ;Color Compare register index in GC
GRAPHICS_MODE           EQU   5     ;Graphics Mode register index in GC
BIT_MASK          EQU   8           ;Bit Mask register index in GC
;
```

```
code        segment     word 'CODE'
            assume      cs:code
Start       proc        near
            cld
;
; Select graphics mode 10h.
;
      mov   ax,10h
      int   10h
;
; Fill the screen with blue.
;
      mov   al,1                    ;blue is color 1
      call  SelectSetResetColor     ;set to draw in blue
      mov   ax,VGA_SEGMENT
      mov   es,ax
      sub   di,di
      mov   cx,7000h
      rep   stosb       ;the value written actually doesn't
                        ; matter, since set/reset is providing
                        ; the data written to display memory
;
; Draw a vertical yellow line.
;
      mov   al,14               ;yellow is color 14
      call  SelectSetResetColor ;set to draw in yellow
      mov   dx,GC_INDEX
      mov   al,BIT_MASK
      out   dx,al               ;point GC Index to Bit Mask
      inc   dx                  ;point to GC Data
      mov   al,10h
      out   dx,al               ;set Bit Mask to 10h
      mov   di,40               ;start in the middle of the top line
      mov   cx,350              ;do full height of screen
VLineLoop:
      mov   al,es:[di]          ;load the latches
      stosb                     ;write next pixel of yellow line (set/reset
                                ; provides the data written to display
                                ; memory, and AL is actually ignored)
      add   di,SCREEN_WIDTH-1   ;point to the next scan line
      loop  VLineLoop
;
; Select write mode 0 and read mode 1.
;
      mov   dx,GC_INDEX
      mov   al,GRAPHICS_MODE
      out   dx,al           ;point GC Index to Graphics Mode register
      inc   dx              ;point to GC Data
      mov   al,00001000b    ;bit 3=1 is read mode 1, bits 1 & 0=00
                            ; is write mode 0
      out   dx,al           ;set Graphics Mode to read mode 1,
                            ; write mode 0
;
; Draw a horizontal green line, one pixel at a time, from left
; to right until color compare reports a yellow pixel is encountered.
;
; Draw in green.
;
      mov   al,2                ;green is color 2
      call  SelectSetResetColor ;set to draw in green
```

```
;
; Set color compare to look for yellow.
;
      mov   dx,GC_INDEX
      mov   al,COLOR_COMPARE
      out   dx,al       ;point GC Index to Color Compare register
      inc   dx          ;point to GC Data
      mov   al,14       ;we're looking for yellow, color 14
      out   dx,al       ;set color compare to look for yellow
      dec   dx          ;point to GC Index
;
; Set up for quick access to Bit Mask register.
;
      mov   al,BIT_MASK
      out   dx,al       ;point GC Index to Bit Mask register
      inc   dx          ;point to GC Data
;
; Set initial pixel mask and display memory offset.
;
      mov   al,80h                ;initial pixel mask
      mov   di,100*SCREEN_WIDTH
                                  ;start at left edge of scan line 100
HLineLoop:
      mov   ah,es:[di]  ;do a read mode 1 (color compare) read.
                        ; This also loads the latches.
      and   ah,al       ;is the pixel of current interest yellow?
      jnz   WaitKeyAndDone    ;yes-we've reached the yellow line, so we're
                        ; done
      out   dx,al       ;set the Bit Mask register so that we
                        ; modify only the pixel of interest
      mov   es:[di],al  ;draw the pixel. The value written is
                        ; irrelevant, since set/reset is providing
                        ; the data written to display memory
      ror   al,1        ;shift pixel mask to the next pixel
      adc   di,0        ;advance the display memory offset if
                        ; the pixel mask wrapped
;
; Slow things down a bit for visibility (adjust as needed).
;
      mov   cx,0
DelayLoop:
      loop  DelayLoop

      jmp   HLineLoop
;
; Wait for a key to be pressed to end, then return to text mode and
; return to DOS.
;
WaitKeyAndDone:
WaitKeyLoop:
      mov   ah,1
      int   16h
      jz    WaitKeyLoop
      sub   ah,ah
      int   16h         ;clear the key
      mov   ax,3
      int   10h         ;return to text mode
      mov   ah,4ch
      int   21h         ;done
Start endp
```

```
;
; Enables set/reset for all planes, and sets the set/reset color
; to AL.
;
SelectSetResetColor    proc  near
      mov   dx,GC_INDEX
      push  ax          ;preserve color
      mov   al,SET_RESET
      out   dx,al       ;point GC Index to Set/Reset register
      inc   dx          ;point to GC Data
      pop   ax          ;get back color
      out   dx,al       ;set Set/Reset register to selected color
      dec   dx          ;point to GC Index
      mov   al,ENABLE_SET_RESET
      out   dx,al       ;point GC Index to Enable Set/Reset register
      inc   dx          ;point to GC Data
      mov   al,0fh
      out   dx,al       ;enable set/reset for all planes
      ret
SelectSetResetColor    endp
code  ends
      end   Start
```

When all Planes "Don't Care"

Still and all, there aren't all that many uses for basic color compare operations. There is, however, a genuinely odd application of read mode 1 that's worth knowing about; but in order to understand that, we must first look at the "don't care" aspect of color compare operation.

As described earlier, during read mode 1 reads the color stored in the Color Compare register is compared to each of the 8 pixels at a given address in VGA memory. But—and it's a big but—any plane for which the corresponding bit in the Color Don't Care register is a 0 is always considered a color compare match, regardless of the values of that plane's bits in the pixels and in the Color Compare register.

Let's look at this another way. A given pixel is controlled by four bits, one in each plane. Normally (when the Color Don't Care register is 0FH), the color in the Color Compare register is compared to the four bits of each pixel; bit 0 of the Color Compare register is compared to the plane 0 bit of each pixel, bit 1 of the Color Compare register is compared to the plane 1 bit of each pixel, and so on. That is, when the lower four bits of the Color Don't Care register are all set to 1, then all four bits of a given pixel must match the Color Compare register in order for a read mode 1 read to return a 1 for that pixel to the CPU.

However, if any bit of the Color Don't Care register is 0, then the corresponding bit of each pixel is unconditionally considered to match the corresponding bit of the Color Compare register. You might think of the Color Don't Care register as selecting exactly which planes should matter in a given read mode 1 read. At the extreme, if all bits of the Color Don't Care register are 0, then read mode 1 reads will always return 0FFH, since all planes are considered to match all bits of all pixels.

Now, we're all prone to using tools the "right" way—that is, in the way in which they were intended to be used. By that token, the Color Don't Care register is clearly intended to mask one or more planes out of a color comparison, and as such, has limited use. However, the Color Don't Care register becomes far more interesting in exactly the "extreme" case described above, where all planes become "don't care" planes.

Why? Well, as I've said, when all planes are "don't care" planes, read mode 1 reads always return 0FFH. Now, when you AND any value with 0FFH, the value remains unchanged, and that can be awfully handy when you're using the bit mask to modify selected pixels in VGA memory. Recall that you must always read VGA memory to load the latches before writing to VGA memory when you're using the bit mask. Traditionally, two separate instructions—a read followed by a write—are used to perform this task. The code in Listing 28.2 uses this approach. Suppose, however, that you've set the VGA to read mode 1, with the Color Don't Care register set to 0 (meaning all reads of VGA memory will return 0FFH). Under these circumstances, you can use a single **AND** instruction to both read and write VGA memory, since ANDing any value with 0FFH leaves that value unchanged.

Listing 28.3 illustrates an efficient use of write mode 3 in conjunction with read mode 1 and a Color Don't Care register setting of 0. The mask in AL is passed directly to the VGA's bit mask (that's how write mode 3 works—see Chapter 4 for details). Because the VGA always returns 0FFH, the single **AND** instruction loads the latches, and writes the value in AL, unmodified, to the VGA, where it is used to generate the bit mask. This is more compact and register-efficient than using separate instructions to read and write, although it is not necessarily faster by cycle count, because on a 486 or a Pentium **MOV** is a 1-cycle instruction, but **AND** with memory is a 3-cycle instruction. However, given display memory wait states, it is often the case that the two approaches run at the same speed, and the register that the above approach frees up can frequently be used to save one or more cycles in any case.

By the way, Listing 28.3 illustrates how write mode 3 can make for excellent pixel- and line-drawing code.

LISTING 28.3 L28-3.ASM

```
; Program that draws a diagonal line to illustrate the use of a
; Color Don't Care register setting of 0FFH to support fast
; read-modify-write operations to VGA memory in write mode 3 by
; drawing a diagonal line.
;
; Note: Works on VGAs only.
;
; By Michael Abrash
;
stack segment     word stack 'STACK'
      db    512 dup (?)
stack ends
;
VGA_SEGMENT       EQU   0a000h
SCREEN_WIDTH      EQU   80    ;in bytes
```

```
GC_INDEX          EQU   3ceh  ;Graphics Controller Index register
SET_RESET         EQU   0     ;Set/Reset register index in GC
ENABLE_SET_RESET  EQU   1     ;Enable Set/Reset register index in GC
GRAPHICS_MODE     EQU   5     ;Graphics Mode register index in GC
COLOR_DONT_CARE   EQU   7     ;Color Don't Care register index in GC
;
code  segment     word 'CODE'
      assume      cs:code
Start proc  near
;
; Select graphics mode 12h.
;
      mov   ax,12h
      int   10h
;
; Select write mode 3 and read mode 1.
;
      mov   dx,GC_INDEX
      mov   al,GRAPHICS_MODE
      out   dx,al
      inc   dx
      in    al,dx             ;VGA registers are readable, bless them!
      or    al,00001011b      ;bit 3=1 selects read mode 1, and
                              ; bits 1 & 0=11 selects write mode 3
      jmp   $+2               ;delay between IN and OUT to same port
      out   dx,al
      dec   dx
;
; Set up set/reset to always draw in white.
;
      mov   al,SET_RESET
      out   dx,al
      inc   dx
      mov   al,0fh
      out   dx,al
      dec   dx
      mov   al,ENABLE_SET_RESET
      out   dx,al
      inc   dx
      mov   al,0fh
      out   dx,al
      dec   dx
;
; Set Color Don't Care to 0, so reads of VGA memory always return 0FFH.
;
      mov   al,COLOR_DONT_CARE
      out   dx,al
      inc   dx
      sub   al,al
      out   dx,al
;
; Set up the initial memory pointer and pixel mask.
;
      mov   ax,VGA_SEGMENT
      mov   ds,ax
      sub   bx,bx
      mov   al,80h
;
; Draw 400 points on a diagonal line sloping down and to the right.
;
```

```
      mov   cx,400
DrawDiagonalLoop:
      and   [bx],al     ;reads display memory, loading the latches,
                        ; then writes AL to the VGA. AL becomes the
                        ; bit mask, and set/reset provides the
                        ; actual data written
      add   bx,SCREEN_WIDTH
                        ; point to the next scan line
      ror   al,1        ;move the pixel mask one pixel to the right
      adc   bx,0        ;advance to the next byte if the pixel mask wrapped
      loop  DrawDiagonalLoop
;
; Wait for a key to be pressed to end, then return to text mode and
; return to DOS.
;
WaitKeyLoop:
      mov   ah,1
      int   16h
      jz    WaitKeyLoop
      sub   ah,ah
      int   16h         ;clear the key
      mov   ax,3
      int   10h         ;return to text mode
      mov   ah,4ch
      int   21h         ;done
Start endp
code  ends
      end   Start
```

I hope I've given you a good feel for what color compare mode is and what it might be used for. Color compare mode isn't particularly easy to understand, but it's not that complicated in actual operation, and it's certainly useful at times; take some time to study the sample code and perform a few experiments of your own, and you may well find useful applications for color compare mode in your graphics code.

A final note: The Read Map register has no effect in read mode 1, and the Color Compare and Color Don't Care registers have no effect either in read mode 0 or when writing to VGA memory. And with that, by gosh, we're actually done with the basics of accessing VGA memory!

Not to worry—that still leaves us a slew of interesting VGA topics, including smooth panning and scrolling, the split screen, color selection, page flipping, and Mode X. And that's not to mention actual uses to which the VGA's hardware can be put, including lines, circles, polygons, and my personal favorite, animation. We've covered a lot of challenging and rewarding ground—and we've only just begun.