Saving Screens and Other VGA Mysteries
Useful Nuggets from the VGA Zen File

There are a number of VGA graphics topics that aren’t quite involved enough to warrant their own chapters, yet still cause a fair amount of programmer headscratching—and thus deserve treatment somewhere in this book. This is the place, and during the course of this chapter we’ll touch on saving and restoring 16-color EGA and VGA screens, the 16-out-of-64 colors issue, and techniques involved in reading and writing VGA control registers.

That’s a lot of ground to cover, so let’s get started!

Saving and Restoring EGA and VGA Screens

The memory architectures of EGAs and VGAs are similar enough to treat both together in this regard. The basic principle for saving EGA and VGA 16-color graphics screens is astonishingly simple: Write each plane to disk separately. Let’s take a look at how this works in the EGA’s hi-res mode 10H, which provides 16 colors at 640×350.

All we need do is enable reads from plane 0 and write the 28,000 bytes of plane 0 that are displayed in mode 10H to disk, then enable reads from plane 1 and write the displayed portion of that plane to disk, and so on for planes 2 and 3. The result is a file that’s 112,000 (28,000 * 4) bytes long, with the planes stored as four distinct 28,000-byte blocks, as shown in Figure 29.1.

The program shown later on in Listing 29.1 does just what I’ve described here, putting the screen into mode 10H, putting up some bit-mapped text so there is something
saving EGA/VGA display memory.

Figure 29.1

to save, and creating the 112K file SNAPSHOT.SCR, which contains the visible portion of the mode 10H frame buffer.

The only part of Listing 29.1 that’s even remotely tricky is the use of the Read Map register (Graphics Controller register 4) to make each of the four planes of display memory readable in turn. The same code is used to write 28,000 bytes of display memory to disk four times, and 28,000 bytes of memory starting at A000:0000 are written to disk each time; however, a different plane is read each time, thanks to the changing setting of the Read Map register. (If this is unclear, refer back to Figure 29.1; you may also want to reread Chapter 28 to brush up on the operation of the Read Map register in particular and reading EGA and VGA memory in general.)

Of course, we’ll want the ability to restore what we’ve saved, and Listing 29.2 does this. Listing 29.2 reverses the action of Listing 29.1, selecting mode 10H and then loading 28,000 bytes from SNAPSHOT.SCR into each plane of display memory. The Map Mask register (Sequence Controller register 2) is used to select the plane to be written to. If your computer is slow enough, you can see the colors of the text change
as each plane is loaded when Listing 29.2 runs. Note that Listing 29.2 does not itself draw any text, but rather simply loads the bit map saved by Listing 29.1 back into the mode 10H frame buffer.

**LISTING 29.1 L29-1.ASM**

; Program to put up a mode 10h EGA graphics screen, then save it ; to the file SNAPSHOT.SCR.

VGA_SEGMENT equ 0a000h
GC_INDEX equ 3ceh
READ_MAP equ 4
DISPLAYED_SCREEN_SIZE equ (640/8)*350

; stack segment para stack 'STACK'
stack db 512 dup(?)
stack ends

Data segment word 'DATA'
SampleText db 'This is bit-mapped text, drawn in hi-res'
   db 'EGA graphics mode 10h.', Odh, Oah, Oah
   db 'Saving the screen (including this text)...
Filename db 'SNAPSHOT.SCR', O
   ; name of file we're saving to
ErrMsg1 db '*** Couldn''t open SNAPSHOT.SCR ***', Odh, Oah, '$'
ErrMsg2 db '*** Error writing to SNAPSHOT.SCR ***', Odh, Oah, '$'
WaitKeyMsg db Odh, Oah, 'Done. Press any key to end...
   ', Odh, Oah, '$'
Handle dw ?
   ; handle of file we're saving to
Plane db ?
   ; plane being read
Data ends

Code segment
assume cs:Code, ds:Data

Start proc near
   mov ax, Data
   mov ds, ax

; Go to hi-res graphics mode.

   mov ax, 10h
   ; AH = 0 means mode set, AL = 10h selects hi-res graphics mode
   int 10h
   ; BIOS video interrupt

; Put up some text, so the screen isn't empty.

   mov ah, 9
   ; DOS print string function
   mov dx, offset SampleText
   int 21h

; Delete SNAPSHOT.SCR if it exists.

   mov ah, 41h
   ; DOS unlink file function
   mov dx, offset Filename
   int 21h

; Create the file SNAPSHOT.SCR.

   mov ah, 3ch
   ; DOS create file function
mov dx, offset Filename
sub cx, cx ; make it a normal file
int 21h
mov [Handle], ax ; save the handle
jnc SaveTheScreen ; we're ready to save if no error
mov ah, 9 ; DOS print string function
mov dx, offset ErrMsg1
int 21h ; notify of the error
jmp short Done ; done

; Loop through the 4 planes, making each readable in turn and
; writing it to disk. Note that all 4 planes are readable at
; A000:0000; the Read Map register selects which plane is readable
; at any one time.

SaveTheScreen:
  mov [Plane], 0 ; start with plane 0

SaveLoop:
  mov cx, GC_INDEX
  mov al, READ_MAP ; set GC Index to Read Map register
  mov dx, al
  inc dx
  mov al, [Plane] ; get the # of the plane we want
      ; to save
  mov dx, al ; set to read from the desired plane
  mov ah, 40h ; DOS write to file function
  mov bx, [Handle]
  mov cx, DISPLAYED_SCREEN_SIZE ; # of bytes to save
  sub dx, dx ; write all displayed bytes at A000:0000
  push ds
  mov si, VGA_SEGMENT
  mov bx, [VGA_SEGMENT]
  int 21h ; write the displayed portion of this plane
  pop ds
  cmp ax, DISPLAYED_SCREEN_SIZE ; did all bytes get written?
  jz SaveLoopBottom
  mov ah, 9 ; DOS print string function
  mov dx, offset ErrMsg2
  int 21h ; notify about the error
  jmp short DoClose ; and done

SaveLoopBottom:
  mov ax, [Plane] ; point to the next plane
  mov [Plane], al
  cmp al, 3 ; have we done all planes?
  jbe SaveLoop ; no, so do the next plane

; Close SNAPSHOT.SCR.

DoClose:
  mov ah, 3eh ; DOS close file function
  mov bx, [Handle]
  int 21h

; Wait for a keypress.

  mov ah, 9 ; DOS print string function
  mov dx, offset WaitKeyMsg
  int 21h ; prompt
  mov ah, 8 ; DOS input without echo function
; Restore text mode.
;  
;   mov ax,3
;   int 10h
;  
; Done.
;  
; Done:
;  
; Start
;   endp
; Code
;   ends
;   Start

LISTING 29.2  L29-2.ASM
; Program to restore a mode 10h EGA graphics screen from
; the file SNAPSHOT.SCR.
;
VGA_SEGMENT    equ  0a000h                        ;Sequence Controller Index register
SC_INDEX       equ  3c4h                        ;Map Mask register index in SC
MAP_MASK       equ  2
DISPLAYED_SCREEN_SIZE equ (640/8)*350 ;# of displayed bytes per plane in a
                                    ; hi-res graphics screen

stack           segment para stack 'STACK'
                db  512 dup (?)
stack           ends

Data            segment word 'DATA'
Filename        db  'SNAPSHOT.SCR',0            ;name of file we're restoring from
ErrMsg1         db  '*** Couldn''t open SNAPSHOT.SCR ***',Oh,0ah,'$'
ErrMsg2         db  '*** Error reading from SNAPSHOT.SCR ***',Oh,0ah,'$'
WaitKeyMsg      db  0dh, 0ah, 'Done. Press any key to end...',0dh,0ah,'$'
Handle          dw  ? ;handle of file we're restoring from
Plane           db  ? ;plane being written
Data            ends

Code            segment
                assume cs:Code, ds:Data
Start           proc   near
                mov ax,Data
                mov ds,ax
                
                ; Go to hi-res graphics mode.
                ;
                ;   mov ax,10h
                ;   int 10h
                ;   ; hi-res graphics mode
                ;   ; BIOS video interrupt
                ;
                ; Open SNAPSHOT.SCR.
                ;
                ;   mov ah,3dh
                ;   mov dx,offset Filename
                ;   sub al,al
                ;   int 21h
                ;   mov [Handle],ax     ;save the handle
                ;   jnc RestoreTheScreen ;we're ready to restore if no error
                ;   mov ah,9
                ;   ;DOS print string function

Saving Screens and Other VGA Mysteries  545
mov dx, offset ErrMsg1
int 21h ; notify of the error
jmp short Done ; and done

; Loop through the 4 planes, making each writable in turn and
; reading it from disk. Note that all 4 planes are writable at
; A000:0000; the Map Mask register selects which planes are readable
; at any one time. We only make one plane readable at a time.

RestoreTheScreen:
mov [Plane].0

RestoreLoop:
mov dx, SC_INDEX
mov al, MAP_MASK
out dx, al
inc dx
mov cl, [Plane]

mov al, 1
shl al, cl
out dx, al
mov ah, 3fh
mov bx, [Handle]
mov cx, DISPLAYED_SCREEN_SIZE
sub dx, dx
push ds
mov si, VGA_SEGMENT
mov ds, si
int 21h ; read the displayed portion of this plane

pop ds
jc ReadError
cmp ax, DISPLAYED_SCREEN_SIZE
jz RestoreLoopBottom

ReadError:
mov ah, 9
mov dx, offset ErrMsg2
int 21h ; notify about the error
jmp short DoClose

RestoreLoopBottom:
mov al, [Plane]
inc ax
mov [Plane], al
cmp al, 3
jbe RestoreLoop

; Close SNAPSHOT.SCR.

DoClose:
mov ah, 3eh
mov bx, [Handle]
int 21h

; Wait for a keypress.

mov ah, 8
int 21h

; Restore text mode.
If you compare Listings 29.1 and 29.2, you will see that the Map Mask register setting used to load a given plane does not match the Read Map register setting used to read that plane. This is so because while only one plane can ever be read at a time, anywhere from zero to four planes can be written to at once; consequently, Read Map register settings are plane selections from 0 to 3, while Map Mask register settings are plane masks from 0 to 15, where a bit 0 setting of 1 enables writes to plane 0, a bit 1 setting of 1 enables writes to plane 1, and so on. Again, Chapter 28 provides a detailed explanation of the differences between the Read Map and Map Mask registers.

Screen saving and restoring is pretty simple, eh? There are a few caveats, of course, but nothing serious. First, the adapter’s registers must be programmed properly in order for screen saving and restoring to work. For screen saving, you must be in read mode 0; if you’re in color compare mode, there’s no telling what bit pattern you’ll save, but it certainly won’t be the desired screen image. For screen restoring, you must be in write mode 0, with the Bit Mask register set to OFFH and Data Rotate register set to 0 (no data rotation and the logical function set to pass the data through unchanged).

While these requirements are no problem if you’re simply calling a subroutine in order to save an image from your program, they pose a considerable problem if you’re designing a hot-key operated TSR that can capture a screen image at any time. With the EGA specifically, there’s never any way to tell what state the registers are currently in, since the registers aren’t readable. (More on this issue later in this chapter.) As a result, any TSR that sets the Bit Mask to OFFH, the Data Rotate register to 0, and so on runs the risk of interfering with the drawing code of the program that’s already running.

What’s the solution? Frankly, the solution is to get VGA-specific. A TSR designed for the VGA can simply read out and save the state of the registers of interest, program those registers as needed, save the screen image, and restore the original settings. From a programmer’s perspective, readable registers are certainly near the top of the list of things to like about the VGA! The remaining installed base of EGAs is steadily dwindling, and you may be able to ignore it as a market today, as you couldn’t even a year or two ago.
If you are going to write a hi-res VGA version of the screen capture program, be sure to account for the increased size of the VGA's mode 12H bit map. The mode 12H (640×480) screen uses 37.5K per plane of display memory, so for mode 12H the displayed screen size equate in Listings 29.1 and 29.2 should be changed to:

```assembly
DISPLAYED_SCREEN_SIZE equ (640/8)*480
```

Similarly, if you're capturing a graphics screen that starts at an offset other than 0 in the segment at A000H, you must change the memory offset used by the disk functions to match. You can, if you so desire, read the start offset of the display memory providing the information shown on the screen from the Start Address registers (CRT Controller registers 0CH and 0DH); these registers are readable even on an EGA.

Finally, be aware that the screen capture and restore programs in Listings 29.1 and 29.2 are only appropriate for EGA/VGA modes 0DH, 0EH, 0FH, 010H, and 012H, since they assume a four-plane configuration of EGA/VGA memory. In all text modes and in CGA graphics modes, and in VGA modes 11H and 13H as well, display memory can simply be written to disk and read back as a linear block of memory, just like a normal array.

While Listings 29.1 and 29.2 are written in assembly, the principles they illustrate apply equally well to high-level languages. In fact, there’s no need for any assembly at all when saving an EGA/VGA screen, as long as the high-level language you're using can perform direct port I/O to set up the adapter and can read and write display memory directly.

---

One tip if you're saving and restoring the screen from a high-level language on an EGA, though: After you've completed the save or restore operation, be sure to put any registers that you've changed back to their default settings. Some high-level languages (and the BIOS as well) assume that various registers are left in a certain state, so on the EGA it's safest to leave the registers in their most likely state. On the VGA, of course, you can just read the registers out before you change them, then put them back the way you found them when you're done.

### 16 Colors out of 64

How does one produce the 64 colors from which the 16 colors displayed by the EGA can be chosen? The answer is simple enough: There’s a BIOS function that lets you select the mapping of the 16 possible pixel values to the 64 possible colors. Let’s lay out a bit of background before proceeding, however.

The EGA sends pixel information to the monitor on 6 pins. This means that there are 2 to the 6th, or 64 possible colors that an EGA can generate. However, for compatibility with pre-EGA monitors, in 200-scan-line modes Enhanced Color Display-compatible monitors ignore two of the signals. As a result, in CGA-compatible modes (modes 4, 5, 6, and the 200-scan-line versions of modes 0, 1, 2, and 3) you can select from only 16 colors (although the colors can still be remapped, as described below). If you’re not hooked up to a monitor capable of displaying 350 scan lines (such as the old...
IBM Color Display), you can never select from more than 16 colors, since those monitors only accept four input signals. For now, we’ll assume we’re in one of the 350-scan line color modes, a group which includes mode 10H and the 350-scan-line versions of modes 0, 1, 2, and 3.

Each pixel comes out of memory (or, in text mode, out of the attribute-handling portion of the EGA) as a 4-bit value, denoting 1 of 16 possible colors. In graphics modes, the 4-bit pixel value is made up of one bit from each plane, with 8 pixels’ worth of data stored at any given byte address in display memory. Normally, we think of the 4-bit value of a pixel as being that pixel’s color, so a pixel value of 0 is black, a pixel value of 1 is blue, and so on, as if that’s a built-in feature of the EGA.

Actually, though, the correspondence of pixel values to color is absolutely arbitrary, depending solely on how the color-mapping portion of the EGA containing the palette registers is programmed. If you cared to have color 0 be bright red and color 1 be black, that could easily be arranged, as could a mapping in which all 16 colors were yellow. What’s more, these mappings affect text-mode characters as readily as they do graphics-mode pixels, so you could map text attribute 0 to white and text attribute 15 to black to produce a black on white display, if you wished.

Each of the 16 palette registers stores the mapping of one of the 16 possible 4-bit pixel values from memory to one of 64 possible 6-bit pixel values to be sent to the monitor as video data, as shown in Figure 29.2. A 4-bit pixel value of 0 causes the 6-bit value

<table>
<thead>
<tr>
<th>Palette Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td>D</td>
</tr>
<tr>
<td>E</td>
</tr>
<tr>
<td>F</td>
</tr>
</tbody>
</table>

4 bits per pixel from display memory or from a text attribute, used to look up a palette register.

6 bits per pixel to the display, from the palette register selected by the 4-bit pixel value.

Color translation via the palette registers.

**Figure 29.2**
stored in palette register 0 to be sent to the display as the color of that pixel, a pixel value of 1 causes the contents of palette register 1 to be sent to the display, and so on. Since there are only four input bits, it stands to reason that only 16 colors are available at any one time; since there are six output bits, however, those 16 colors can be mapped to any of 64 colors. The mapping for each of the 16 pixel values is controlled by the lower six bits of the corresponding palette register, as shown in Figure 29.3. Secondary red, green, and blue are less-intense versions of red, green, and blue, although their exact effects vary from monitor to monitor. The best way to figure out what the 64 colors look like on your monitor is to see them, and that's just what the program in Listing 29.3, which we'll discuss shortly, lets you do.

How does one go about setting the palette registers? Well, it's certainly possible to set the palette registers directly by addressing them at registers 0 through 0FH of the Attribute Controller. However, setting the palette registers is a bit tricky—bit 5 of the Attribute Controller Index register must be 0 while the palette registers are written to, and glitches can occur if the updating doesn't take place during the blanking interval—and besides, it turns out that there's no need at all to go straight to the hardware on this one. Conveniently, the EGA BIOS provides us with video function 10H, which supports setting either any one palette register or all 16 palette registers (and the overscan register as well) with a single video interrupt.

Video function 10H is invoked by performing an INT 10H with AH set to 10H. If AL is 0 (subfunction 0), then BL contains the number of the palette register to set, and BH contains the value to set that register to. If AL is 1 (subfunction 1), then BH contains the value to set the overscan (border) color to. Finally, if AL is 2 (subfunction 2), then ES:DX points to a 17-byte array containing the values to set palette registers 0-15 and the overscan register to. (For completeness, although it's unrelated to the palette registers, there is one more subfunction of video function 10H. If AL = 3

<table>
<thead>
<tr>
<th>Palette Register</th>
<th>R'</th>
<th>G'</th>
<th>B'</th>
<th>R</th>
<th>G</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

- R' = secondary red
- G' = secondary green
- B' = secondary blue
- R = red
- G = green
- B = blue

*Bit organization within a palette register.*

**Figure 29.3**
(subfunction 3), bit 0 of BL is set to 1 to cause bit 7 of text attributes to select blinking, or set to 0 to cause bit 7 of text attributes to select high-intensity reverse video.)

Listing 29.3 uses video function 10H, subfunction 2 to step through all 64 possible colors. This is accomplished by putting up 16 color bars, one for each of the 16 possible 4-bit pixel values, then changing the mapping provided by the palette registers to select a different group of 16 colors from the set of 64 each time a key is pressed. Initially, colors 0-15 are displayed, then 1-16, then 2-17, and so on up to color 3FH wrapping around to colors 0-14, and finally back to colors 0-15. (By the way, at mode set time the 16 palette registers are not set to colors 0-15, but rather to 0H, 1H, 2H, 3H, 4H, 5H, 14H, 7H, 38H, 39H, 3AH, 3BH, 3CH, 3DH, 3EH, and 3FH, respectively. Bits 6, 5, and 4—secondary red, green, and blue—are all set to 1 in palette registers 8-15 in order to produce high-intensity colors. Palette register 6 is set to 14H to produce brown, rather than the yellow that the expected value of 6H would produce.)

When you run Listing 29.3, you'll see that the whole screen changes color as each new color set is selected. This occurs because most of the pixels on the screen have a value of 0, selecting the background color stored in palette register 0, and we're reprogramming palette register 0 right along with the other 15 palette registers.

It's important to understand that in Listing 29.3 the contents of display memory are never changed after initialization. The only change is the mapping from the 4-bit pixel data coming out of display memory to the 6-bit data going to the monitor. For this reason, it's technically inaccurate to speak of bits in display memory as representing colors; more accurately, they represent attributes in the range 0-15, which are mapped to colors 0-3FH by the palette registers.

**LISTING 29.3 L29-3.ASM**

; Program to illustrate the color mapping capabilities of the
; EGA's palette registers.

VGA_SEGMENT equ 0a000h
SC_INDEX equ 3c4h ;Sequence Controller Index register
MAP_MASK equ 2 ;Map Mask register index in SC
BAR_HEIGHT equ 14 ;height of each bar
TOP_BAR equ BAR_HEIGHT*6 ;start the bars down a bit to

stack segment para stack 'STACK'

Data segment word 'DATA'

KeyMsg db 'Press any key to see the next color set. '
db 'There are 64 color sets in all.'
db 0dh, 0ah, 0ah, 0ah, 0ah

; Used to label the attributes of the color bars.

Saving Screens and Other VGA Mysteries 551
AttributeNumbers label byte
x- 0
rept 16
if x lt 10
  db '0', x+'0', 'h', 0ah, 8, 8, 8
else
  db '0', x+'A'-10, 'h', 0ah, 8, 8, 8
endif
x- x+1
endm
db 'S'

; Used to label the colors of the color bars. (Color values are filled in on the fly.)

ColorNumbers label byte
rept 16
db '000h', 0ah, 8, 8, 8, 8
endm
COLOR_ENTRY_LENGTH equ ($-ColorNumbers)/16
db 'S'

; CurrentColor db ?

; Space for the array of 16 colors we'll pass to the BIOS, plus an overscan setting of black.

ColorTable db 16 dup (?), 0

Data ends

Code segment
assume cs:Code, ds:Data
Start proc near
cld
mov ax, Data
mov ds, ax

; Go to hi-res graphics mode.

mov ax, 10h ;AH = 0 means mode set, AL = 10h selects hi-res graphics mode
int 10h ;BIOS video interrupt

; Put up relevant text.

mov ah, 9 ;DOS print string function
mov dx, offset KeyMsg
int 21h

; Put up the color bars, one in each of the 16 possible pixel values (which we'll call attributes).

mov cx, 16 ;we'll put up 16 color bars
sub al, a1 ;start with attribute 0
BarLoop:
push ax
push cx
call BarUp
pop cx
pop ax
inc ax ;select the next attribute
loop BarLoop
;
; Put up the attribute labels.
;
mov ah,2 ;video interrupt set cursor position function
sub bh, bh ;page 0
mov dh, TOP_BAR/14 ;counting in character rows, match to
; top of first bar, counting in
; scan lines
mov dl,16 ;just to left of bars
int 10h
mov ah,9 ;DOS print string function
mov dx,offset AttributeNumbers
int 21h
;
; Loop through the color set, one new setting per keypress.
;
mov [CurrentColor],0 ;start with color zero
ColorLoop:
;
Set the palette registers to the current color set, consisting
of the current color mapped to attribute 0, current color + 1
mapped to attribute 1, and so on.
;
mov al,[CurrentColor]
mov bx,offset ColorTable
mov cx,16 ;we have 16 colors to set
PaletteSetLoop:
and al,3fh ;limit to 6-bit color values
mov [bx].al ;build the 16-color table used for setting
inc bx ; the palette registers
inc ax
loop PaletteSetLoop
mov ah,10h ;video interrupt palette function
mov al,2 ;subfunction to set all 16 palette registers
; and overscan at once
mov dx,offset ColorTable
push ds
pop es ;ES:DX points to the color table
int 10h ;invoke the video interrupt to set the palette
;
; Put up the color numbers, so we can see how attributes map
; to color values, and so we can see how each color # looks
; (at least on this particular screen).
;
call ColorNumbersUp
;
; Wait for a keypress, so they can see this color set.
;
WaitKey:
mov ah,8 ;DOS input without echo function
int 21h
;
; Advance to the next color set.
;
mov al,[CurrentColor]
inc ax
mov [CurrentColor].al
cmp al,64
jbe ColorLoop

Saving Screens and Other VGA Mysteries 553
; : Restore text mode.
; 
; mov ax,3
; int 10h
;
; : Done.
;
; Done:
; mov ah,4ch ;DOS terminate function
; int 21h
;
; : Puts up a bar consisting of the specified attribute (pixel value),
; at a vertical position corresponding to the attribute.
;
; : Input: AL = attribute
;
; BarUp proc near
; mov dx,SC_INDEX
; mov ah,al
; mov al,MAP_MASK
; out dx,al
; inc dx
; mov al,ah
; out dx,al
; mov ah,BAR_HEIGHT
; mov al,ah
; set the Map Mask register to produce
; the desired color
; mov ah,BAR_HEIGHT
; mov al,BAR_HEIGHT
; row of top of bar
; mov dx,80
; mul dx
; rows are 80 bytes long
; add ax,20
; offset in bytes of start of scan line bar
; starts on
; mov di,ax
; mov ax,VGA_SEGMENT
; mov es,ax
; ES:DI points to offset of upper left
; corner of bar
; mov dx,BAR_HEIGHT
; mov al,0ffh
; BarLineLoop:
; mov cx,40
; make the bars 40 wide
; rep stosb
; do one scan line of the bar
; add di,40
; point to the start of the next scan line
; of the bar
; dec dx
; jnz BarLineLoop
; BarUp
; ret
; endp
;
; : Converts AL to a hex digit in the range 0-F.
;
; BinToHexDigit proc near
; cmp al,9
; ja IsHex
; add al,'0'
; ret
;
; IsHex:
; add al,'A'-10
; ret
; BinToHexDigit endp
; Displays the color values generated by the color bars given the 
; current palette register settings off to the right of the color 
; bars.

ColorNumbersUp proc near

mov ah,2 ; video interrupt set cursor position function
sub bh,bh ; page 0
mov dh,TOP_BAR14 ; counting in character rows, match to 
; top of first bar, counting in 
; scan lines
mov dl,20+40+1 ; just to right of bars
int 10h
mov al,[CurrentColor]; start with the current color
mov bx,offset ColorNumbers+1
mov cx,16 ; we've got 16 colors to do

ColorNumberLoop:
push ax ; save the color #
and ax,3fh ; limit to 6-bit color values
shr ax,1
shr ax,1
shr ax,1
shr ax,1
; isolate the high nibble of the color #
call BinToHexDigit ; convert the high color # nibble
mov [bx],al ; and put it into the text
pop ax ; get back the color #
push ax ; save the color #
and ax,0fh ; isolate the low color # nibble
call BinToHexDigit ; convert the low nibble of the 
; color # to ASCII
mov [bx+1],al ; and put it into the text
add bx, COLOR_ENTRY_LENGTH ; point to the next entry
pop ax ; get back the color #
inc ax ; next color #
loop ColorNumberLoop
mov ah,9 ; DOS print string function
mov dx,offset ColorNumbers
int 21h ; put up the attribute numbers

ret

ColorNumbersUp endp

; Start
Code end Code

Overscan

While we're at it, I'm going to touch on overscan. Overscan is the color of the border 
of the display, the rectangular area around the edge of the monitor that's outside 
the region displaying active video data but inside the blanking area. The overscan 
(or border) color can be programmed to any of the 64 possible colors by either 
setting Attribute Controller register 11H directly or calling video function 10H, 
subfunction 1.
On ECD-compatible monitors, however, there’s too little scan time to display a proper border when the EGA is in 350-scan-line mode, so overscan should always be 0 (black) unless you’re in 200-scan-line mode. Note, though, that a VGA can easily display a border on a VGA-compatible monitor, and VGAs are in fact programmed at mode set for an 8-pixel-wide border in all modes; all you need do is set the overscan color on any VGA to see the border.

A Bonus Blanker

An interesting bonus: The Attribute Controller provides a very convenient way to blank the screen, in the form of the aforementioned bit 5 of the Attribute Controller Index register (at address 3C0H after the Input Status 1 register—3DAH in color, 3BAH in monochrome—has been read and on every other write to 3C0H thereafter). Whenever bit 5 of the AC Index register is 0, video data is cut off, effectively blanking the screen. Setting bit 5 of the AC Index back to 1 restores video data immediately. Listing 29.4 illustrates this simple but effective form of screen blanking.

**LISTING 29.4 L29-4.ASM**

; Program to demonstrate screen blanking via bit 5 of the Attribute Controller Index register.

AC_INDEX equ 3C0H ;Attribute Controller Index register
INPUT_STATUS_1 equ 3DAH ;color-mode address of the Input Status 1 register

; Macro to wait for and clear the next keypress.

WAIT_KEY macro
  mov ah,8 ;DOS input without echo function
  int 21h
endm

;DOS input without echo function

stack segment para stack 'STACK'
stack ends

Data segment word 'DATA'
SampleText db 'This is bit-mapped text, drawn in hi-res'
  db 'EGA graphics mode 10h.', Odh, Oah, Oah
  db 'Press any key to blank the screen, then'
  db 'any key to unblank it.', Odh, Oah
  db 'then any key to end.$'
Data ends

Code segment

Start proc near
  assume cs:Code, ds:Data

mov ax, Data
mov ds, ax

; Go to hi-res graphics mode.

;mov ax,10h ;AH = 0 means mode set, AL = 10h selects hi-res graphics mode
int 10h ;BIOS video interrupt

556 Chapter 29
Does that do it for color selection? Yes and no. For the EGA, we’ve covered the whole of color selection—but not so for the VGA. The VGA can emulate everything we’ve discussed, but actually performs one 4-bit to 8-bit translation (except in 256-color modes, where all 256 colors are simultaneously available), followed by yet another translation, this one 8-bit to 18-bit. What’s more, the VGA has the ability to flip instantly through as many as 16 16-color sets. The VGA’s color selection capabilities, which are supported by another set of BIOS functions, can be used to produce stunning color effects, as we’ll see when we cover them starting in Chapter 33.

Saving Screens and Other VGA Mysteries  557
Modifying VGA Registers

EGA registers are not readable. VGA registers are readable. This revelation will not come as news to most of you, but many programmers still insist on setting entire VGA registers even when they’re modifying only selected bits, as if they were programming the EGA. This comes to mind because I recently received a query inquiring why write mode 1 (in which the contents of the latches are copied directly to display memory) didn’t work in Mode X. (I’ll go into Mode X in detail later in this book.) Actually, write mode 1 does work in Mode X; it didn’t work when this particular correspondent enabled it because he did so by writing the value 01H to the Graphics Mode register. As it happens, the write mode field is only one of several fields in that register, as shown in Figure 29.4. In 256-color modes, one of the other fields—bit 6, which enables 256-color pixel formatting—is not 0, and setting it to 0 messes up the screen quite thoroughly.

The correct way to set a field within a VGA register is, of course, to read the register, mask off the desired field, insert the desired setting, and write the result back to the register. In the case of setting the VGA to write mode 1, do this:

```
mov dx,3ceh ;Graphics controller index
mov al,5 ;Graphics mode reg index
out dx,al ;point GC index to G_MODE
inc dx ;Graphics controller data
in al,dx ;get current mode setting
and al,not 3 ;mask off write mode field
or al,1 ;set write mode field to 1
out dx,al ;set write mode 1
```

This approach is more of a nuisance than simply setting the whole register, but it’s safer. It’s also slower; for cases where you must set a field repeatedly, it might be worthwhile to read and mask the register once at the start, and save it in a variable, so that the value is readily available in memory and need not be repeatedly read from the port. This approach is especially attractive because INS are much slower than memory accesses on 386 and 486 machines.

Astute readers may wonder why I didn’t put a delay sequence, such as JMP $+2, between the IN and OUT involving the same register. There are, after all, guidelines from IBM, specifying that a certain period should be allowed to elapse before a second access to an I/O port is attempted, because not all devices can respond as rapidly as a 286 or faster CPU can access a port. My answer is that while I can’t guarantee that a delay isn’t needed, I’ve never found a VGA that required one; I suspect that the delay specification has more to do with motherboard chips such as the timer, the interrupt controller, and the like, and I sure hate to waste the delay time if it’s not necessary. However, I’ve never been able to find anyone with the definitive word on whether delays might ever be needed when accessing VGAs, so if
you know the gospel truth, or if you know of a VGA/processor combo that does require delays, please let me know by contacting me through the publisher. You’d be doing a favor for a whole generation of graphics programmers who aren’t sure whether they’re skating on thin ice without those legendary delays.