Chapter 31

Higher 256-Color Resolution on the VGA
One of the more appealing features of the VGA is its ability to display 256 simultaneous colors. Unfortunately, one of the less appealing features of the VGA is the limited resolution (320x200) of the one 256-color mode the IBM-standard BIOS supports. (There are, of course, higher resolution 256-color modes in the legion of SuperVGAs, but they are by no means a standard, and differences between seemingly identical modes from different manufacturers can be vexing.) More colors can often compensate for less resolution, but the resolution difference between the 640x480 16-color mode and the 320x200 256-color mode is so great that many programmers must regrettfully decide that they simply can’t afford to use the 256-color mode.

If there’s one thing we’ve learned about the VGA, however, it’s that there’s never just one way to do things. With the VGA, alternatives always exist for the clever programmer, and that’s more true than you might imagine with 256-color mode. Not only is there a high 256-color resolution, there are lots of higher 256-color resolutions, going all the way up to 360x480—and that’s with the vanilla IBM VGA!

In this chapter, I’m going to focus on one of my favorite 256-color modes, which provides 320x400 resolution and two graphics pages and can be set up with very little reprogramming of the VGA. In the next chapter, I’ll discuss higher-resolution 256-color modes, and starting in Chapter 47, I’ll cover the high-performance “Mode X” 256-color programming that many games use.

So. Let’s get started.
Why 320x200? Only IBM Knows for Sure

The first question, of course, is, “How can it be possible to get higher 256-color resolutions out of the VGA?” After all, there were no unused higher resolutions to be found in the CGA, Hercules card, or EGA.

The answer is another question: “Why did IBM not use the higher-resolution 256-color modes of the VGA?” The VGA is easily capable of twice the 200-scan-line vertical resolution of mode 13H, the 256-color mode, and IBM clearly made a decision not to support a higher-resolution 256-color mode. In fact, mode 13H does display 400 scan lines, but each row of pixels is displayed on two successive scan lines, resulting in an effective resolution of 320x200. This is the same scan-doubling approach used by the VGA to convert the CGA’s 200-scan-line modes to 400 scan lines; however, the resolution of the CGA has long been fixed at 200 scan lines, so IBM had no choice with the CGA modes but to scan-double the lines. Mode 13H has no such historical limitation—it’s the first 256-color mode ever offered by IBM, if you don’t count the late and unlamented Professional Graphics Controller (PGC). Why, then, would IBM choose to limit the resolution of mode 13H?

There’s no way to know, but one good guess is that IBM wanted a standard 256-color mode across all PS/2 computers (for which the VGA was originally created), and mode 13H is the highest-resolution 256-color mode that could fill the bill. You see, each 256-color pixel requires one byte of display memory, so a 320x200 256-color mode requires 64,000 bytes of display memory. That’s no problem for the VGA, which has 256K of display memory, but it’s a stretch for the MCGA of the Model 30, since the MCGA comes with only 64K.

On the other hand, the smaller display memory size of the MCGA also limits the number of colors supported in 640x480 mode to 2, rather than the 16 supported by the VGA. In this case, though, IBM simply created two modes and made both available on the VGA: mode 11H for 640x480 2-color graphics and mode 12H for 640x480 16-color graphics. The same could have been done for 256-color graphics—but wasn’t. Why? I don’t know. Maybe IBM just didn’t like the odd aspect ratio of a 320x400 graphics mode. Maybe they didn’t want to have to worry about how to map in more than 64K of display memory. Heck, maybe they made a mistake in designing the chip. Whatever the reason, mode 13H is really a 400-scan-line mode masquerading as a 200-scan-line mode, and we can readily end that masquerade.

320x400 256-Color Mode

Okay, what’s so great about 320x400 256-color mode? Two things: easy, safe mode sets and page flipping.

As I said above, mode 13H is really a 320x400 mode, albeit with each line doubled to produce an effective resolution of 320x200. That means that we don’t need to change any display timings, widths, or heights in order to tweak mode 13H into 320x400
mode—and that makes 320×400 a safe choice. Basically, 320×400 mode differs from mode 13H only in the settings of mode bits, which are sure to be consistent from one VGA clone to the next and which work equally well with all monitors. The other hires 256-color modes differ from mode 13H not only in the settings of the mode bits but also in the settings of timing and dimension registers, which may not be exactly the same on all VGA clones and particularly not on all multisync monitors. (Because multisyncs sometimes shrink the active area of the screen when used with standard VGA modes, some VGAs use alternate register settings for multisync monitors that adjust the CRT Controller timings to use as much of the screen area as possible for displaying pixels.)

The other good thing about 320×400 256-color mode is that two pages are supported. Each 320×400 256-color mode requires 128,000 bytes of display memory, so we can just barely manage two pages in 320×400 mode, one starting at offset 0 in display memory and the other starting at offset 8000H. Those two pages are the largest pair of pages that can fit in the VGA’s 256K, though, and the higher-resolution 256-color modes, which use still larger bitmaps (areas of display memory that control pixels on the screen), can’t support two pages at all. As we’ve seen in earlier chapters and will see again in this book, paging is very useful for off-screen construction of images and fast, smooth animation.

That’s why I like 320×400 256-color mode. The next step is to understand how display memory is organized in 320×400 mode, and that’s not so simple.

Display Memory Organization in 320×400 Mode
First, let’s look at why display memory must be organized differently in 320×400 256-color mode than in mode 13H. The designers of the VGA intentionally limited the maximum size of the bitmap in mode 13H to 64K, thereby limiting resolution to 320×200. This was accomplished in hardware, so there is no way to extend the bitmap organization of mode 13H to 320×400 mode.

That’s a shame, because mode 13H has the simplest bitmap organization of any mode—one long, linear bitmap, with each byte controlling one pixel. We can’t have that organization, though, so we’ll have to find an acceptable substitute if we want to use a higher 256-color resolution.

We’re talking about the VGA, so of course there are actually several bitmap organizations that let us use higher 256-color resolutions than mode 13H. The one I like best is shown in Figure 31.1. Each byte controls one 256-color pixel. Pixel 0 is at address 0 in plane 0, pixel 1 is at address 0 in plane 1, pixel 2 is at address 0 in plane 2, pixel 3 is at address 0 in plane 3, pixel 4 is at address 1 in plane 0, and so on.

Let’s look at this another way. Ideally, we’d like one long bitmap, with each pixel at the address that’s just after the address of the pixel to the left. Well, that’s true in this case too, if you consider the number of the plane that the pixel is in to be part of the pixel’s address. View the pixel numbers on the screen as increasing from left to right.
and from the end of one scan line to the start of the next. Then the pixel number, \( n \), of the pixel at display memory address \( \text{address} \) in plane \( \text{plane} \) is:

\[
n = (\text{address} \times 4) + \text{plane}
\]

To turn that around, the display memory address of pixel number \( n \) is given by

\[
\text{address} = n / 4
\]

and the plane of pixel \( n \) is given by:

\[
\text{plane} = n \mod 4
\]

Basically, the full address of the pixel, its pixel number, is broken into two components: the display memory address and the plane.

By the way, because 320x400 mode has a significantly different memory organization from mode 13H, the BIOS text routines won’t work in 320x400 mode. If you want to draw text in 320x400 mode, you’ll have to look up a font in the BIOS ROM and draw
the text yourself. Likewise, the BIOS read pixel and write pixel routines won't work in 320x400 mode, but that's no problem because I'll provide equivalent routines in the next section.

Our next task is to convert standard mode 13H into 320x400 mode. That's accomplished by undoing some of the mode bits that are set up especially for mode 13H, so that from a programming perspective the VGA reverts to a straightforward planar model of memory. That means taking the VGA out of chain 4 mode and doubleword mode, turning off the double display of each scan line, making sure chain mode, odd/even mode, and word mode are turned off, and selecting byte mode for video data display. All that's done in the Set320×400Mode subroutine in Listing 31.1, which we'll discuss next.

Reading and Writing Pixels

The basic graphics functions in any mode are functions to read and write single pixels. Any more complex function can be built on these primitives, although that's rarely the speediest solution. What's more, once you understand the operation of the read and write pixel functions, you've got all the knowledge you need to create functions that perform more complex graphics functions. Consequently, we'll start our exploration of 320x400 mode with pixel-at-a-time line drawing.

Listing 31.1 draws 8 multicolored octagons in turn, drawing a new one on top of the old one each time a key is pressed. The main-loop code of Listing 31.1 should be easily understood; a series of diagonal, horizontal, and vertical lines are drawn one pixel at a time based on a list of line descriptors, with the draw colors incremented for each successive time through the line list.

LISTING 31.1 L31-1.ASM

; Program to demonstrate pixel drawing in 320x400 256-color mode on the VGA. Draws 8 lines to form an octagon, a pixel at a time. Draws 8 octagons in all, one on top of the other, each in a different color set. Although it's not used, a pixel read function is also provided.

VGA_SEGMENT equ Oa000h
SC_INDEX equ 3c4h ; Sequence Controller Index register
GC_INDEX equ 3ceh ; Graphics Controller Index register
CRTC_INDEX equ 3d4h ; CRT Controller Index register
MAP_MASK equ 2 ; Map Mask register index in SC
MEMORY_MODE equ 4 ; Memory Mode register index in SC
MAX_SCAN_LINE equ 9 ; Maximum Scan Line reg index in CRTC
START_ADDRESS_HIGH equ 0ch ; Start Address High reg index in CRTC
UNDERLINE equ 14h ; Underline Location reg index in CRTC
MODE_CONTROL equ 17h ; Mode Control register index in CRTC
READ_MAP equ 4 ; Read Map register index in GC
GRAPHICS_MODE equ 5 ; Graphics Mode register index in GC
MISCELLANEOUS equ 6 ; Miscellaneous register index in GC
SCREEN_WIDTH equ 320 ; # of pixels across screen
SCREEN_HEIGHT equ 400 ; # of scan lines on screen
WORD_OUTS_OK equ 1 ; set to 0 to assemble for computers that can't handle word outs to indexed VGA registers

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

Data segment word 'DATA'

BaseColor db 0

; Structure used to control drawing of a line.

LineControl struc
StartX dw ?
StartY dw ?
LineXInc dw ?
LineYInc dw ?
BaseLength dw ?
LineColor db ?
LineControl ends

; List of descriptors for lines to draw.

LineList label LineControl
LineControl<130,110,1,0,60,0>
LineControl<190,110,1,1,60,1>
LineControl<250,170,0,1,60,2>
LineControl<250,230,-1,1,60,3>
LineControl<190,290,-1,0,60,4>
LineControl<130,290,-1,-1,60,5>
LineControl<70,230,0,-1,60,6>
LineControl<70,170,1,-1,60,7>
LineControl<0,0,0,0,0>

Data ends

; Macro to output a word value to a port.

OUT_WORD macro
if WORD_OUTS_OK
  out dx,ax
else
  out dx,al
  inc dx
  xchg ah,al
  out dx,al
  dec dx
  xchg ah,al
endif
endm

; Macro to output a constant value to an indexed VGA register.

CONSTANT_TO_INDEXED_REGISTER macro ADDRESS, INDEX, VALUE
  mov dx,ADDRESS
  mov ax,(VALUE shl 8) + INDEX
  OUT_WORD
  endm

594 Chapter 31
; Sets up 320x400 256-color modes.
; Input: none
; Output: none

Higher 256-Color Resolution on the VGA 595
Set320By400Mode proc near
    ; First, go to normal 320x200 256-color mode, which is really a
    ; 320x400 256-color mode with each line scanned twice.
    ;
    ; mov  ax,0013h   ;AH = 0 means mode set, AL = 13h selects
    ;      int 10h     ;BIOS video interrupt
    ;
    ; Change CPU addressing of video memory to linear (not odd/even,
    ; chain, or chain 4), to allow us to access all 256K of display
    ; memory. When this is done, VGA memory will look just like memory
    ; in modes 10h and 12h, except that each byte of display memory will
    ; control one 256-color pixel, with 4 adjacent pixels at any given
    ; address, one pixel per plane.
    ;
    ; mov  dx,SC_INDEX
    ; mov  ax,MEMORY_MODE
    ; out  dx,ax
    ; inc  dx
    ; in   ax,dx
    ; and  ax,not 08h
    ; or  ax,04h
    ; out  dx,ax
    ; mov  dx,GC_INDEX
    ; mov  ax,GRAPHICS_MODE
    ; out  dx,ax
    ; inc  dx
    ; in   ax,dx
    ; and  ax,not 10h
    ; out  dx,ax
    ; dec  dx
    ; mov  ax,MISCELLANEOUS
    ; out  dx,ax
    ; inc  dx
    ; in   ax,dx
    ; and  ax,not 02h
    ; out  dx,ax
    ;
    ; Now clear the whole screen, since the mode 13h mode set only
    ; cleared 64K out of the 256K of display memory. Do this before
    ; we switch the CRTC out of mode 13h, so we don't see garbage
    ; on the screen when we make the switch.
    ;
    ; CONSTANT_TO_INDEXED_REGISTER SC_INDEX,MAP_MASK,0fh
    ;   :enable writes to all planes, so
    ;      we can clear 4 pixels at a time
    ;
    ; mov  ax,VGA_SEGMENT
    ; mov  es,ax
    ; sub  di,di
    ; mov  ax,di
    ; mov  cx,8000h  ;# of words in 64K
    ; cld
    ; rep  stosw     ;clear all of display memory
    ;
    ; Tweak the mode to 320x400 256-color mode by not scanning each
    ; line twice.
    ;
    ; mov  dx,CRTC_INDEX
    ; mov  ax,MAX_SCAN_LINE
    ; out  dx,ax

596 Chapter 31
mov a1,UNDERLINE
out dx,al
inc dx
in al,dx
and al,not 40h ;turn off doubleword
out dx,al
dec dx
mov a1,MODE_CONTROL
out dx,al
inc dx
in al,dx
or al,40h ;turn on the byte mode bit, so memory is
; scanned for video data in a purely
; linear way, just as in modes 10h and 12h
out dx,al
ret
Set320By400Mode endp
;
; Draws a pixel in the specified color at the specified
; location in 320x400 256-color mode.
;
; Input:
; CX = X coordinate of pixel
; DX = Y coordinate of pixel
; BL = pixel color
;
; Output: none
;
; Registers altered: AX, CX, DX, DI, ES
;
WritePixel proc near
mov ax,VGA_SEGMENT
mov es,ax
mov ax,SCREEN_WIDTH/4
mul dx
push cx ;point to display memory
push dx ;point to start of desired row
shr cx,1 ;there are 4 pixels at each address, so
shr cx,1 ;each 320-pixel row is 80 bytes wide
add ax,cx ;in each plane
mov dx,ax
pop cx ;point to the pixel's address
pop dx
shr cx,1 ;there are 4 pixels at each address
shr cx,1 ;so divide the X coordinate by 4
add ax,cx
mov dx,ax
pop cx ;get back the X coordinate
pop dx
shr cx,1 ;get the plane # of the pixel
shr cx,1 ;set aside the X coordinate
add ax,cx
mov ah,01h
shl ah,cl
mov ah,1
shr ah,cl
mov al,MAP_MASK
mov dx,SC_INDEX
OUT_WORD ;set the bit corresponding to the plane
; the pixel is in
mov ah,MAP_MASK
mov al,di
mov dx,SC_INDEX
OUT_WORD ;set to write to the proper plane for
; the pixel
;
Higher 256-Color Resolution on the VGA
mov es:[di],bl :draw the pixel
ret
WritePixel endp
:
; Reads the color of the pixel at the specified location in 320x400
; 256-color mode.
;
; Input:
;  CX - X coordinate of pixel to read
;  DX - Y coordinate of pixel to read
;
; Output:
;  AL - pixel color
;
; Registers altered: AX, CX, DX, SI, ES
;
ReadPixel proc near
mov ax,VGA_SEGMENT
mov es,ax
mov ax,SCREEN_WIDTH/4
:point to display memory
mul dx
:point to start of desired row
push cx
:set aside the X coordinate
shr cx.1
:there are 4 pixels at each address
shr cx.1
: so divide the X coordinate by 4
add ax,cx
:point to the pixel's address
mov si,ax
pop ax
:get back the X coordinate
and ah.3
:get the plane # of the pixel
mov ah,al
mov al,READ_MAP
mov dx,GC_INDEX
OUT_WORD
;set to read from the proper plane for
:the pixel
lods byte ptr es:[si] :read the pixel
ret
ReadPixel endp
:
; Waits for the next key and returns it in AX.
;
; Input: none
;
; Output:
;  AX - full 16-bit code for key pressed
;
GetNextKey proc near
WaitKey:
mov ah,1
int 16h
jz WaitKey ;wait for a key to become available
sub ah,ah
int 16h ;read the key
ret
GetNextKey endp
:
Code ends
:
end Start
The interesting aspects of Listing 31.1 are three. First, the **Set320x400Mode** subroutine selects 320×400 256-color mode. This is accomplished by performing a mode 13H mode set followed by then putting the VGA into standard planar byte mode. **Set320x400Mode** zeros display memory as well. It’s necessary to clear display memory even after a mode 13H mode set because the mode 13H mode set clears only the 64K of display memory that can be accessed in that mode, leaving 192K of display memory untouched.

The second interesting aspect of Listing 31.1 is the **WritePixel** subroutine, which draws a colored pixel at any x,y addressable location on the screen. Although it may not be obvious because I’ve optimized the code a little, the process of drawing a pixel is remarkably simple. First, the pixel’s display memory address is calculated as

\[
\text{address} = (y \times (\text{SCREEN\_WIDTH} / 4)) + (x / 4)
\]

which might be more recognizable as:

\[
\text{address} = (((y \times \text{SCREEN\_WIDTH}) + x) / 4)
\]

(There are 4 pixels at each display memory address in 320×400 mode, hence the division by 4.) Then the pixel’s plane is calculated as

\[
\text{plane} = x \text{ and } 3
\]

which is equivalent to:

\[
\text{plane} = x \text{ modulo } 4
\]

The pixel’s color is then written to the addressed byte in the addressed plane. That’s all there is to it!

The third item of interest in Listing 31.1 is the **ReadPixel** subroutine. **ReadPixel** is virtually identical to **WritePixel**, save that in **ReadPixel** the Read Map register is programmed with a plane number, while **WritePixel** uses a plane mask to set the Map Mask register. Of course, that difference merely reflects a fundamental difference in the operation of the two registers. (If that’s Greek to you, refer back to Chapters 23–30 for a refresher on VGA programming.) **ReadPixel** isn’t used in Listing 31.1, but I’ve included it because, as I said above, the read and write pixel functions together can support a whole host of more complex graphics functions.

How does 320×400 256-color mode stack up as regards performance? As it turns out, the programming model of 320×400 mode is actually pretty good for pixel drawing, pretty much on a par with the model of mode 13H. When you run Listing 31.1, you’ll no doubt notice that the lines are drawn quite rapidly. (In fact, the drawing could be considerably faster still with a dedicated line-drawing subroutine, which would avoid the multiplication associated with each pixel in Listing 31.1.)

In 320×400 mode, the calculation of the memory address is not significantly slower than in mode 13H, and the calculation and selection of the target plane is quickly accomplished. As with mode 13H, 320×400 mode benefits tremendously from the byte-per-pixel organization of 256-color mode, which eliminates the need for the
time-consuming pixel-masking of the 16-color modes. Most important, byte-per-pixel modes never require read-modify-write operations (which can be extremely slow due to display memory wait states) in order to clip and draw pixels. To draw a pixel, you just store its color in display memory—what could be simpler?

More sophisticated operations than pixel drawing are less easy to accomplish in 320×400 mode, but with a little ingenuity it is possible to implement a reasonably efficient version of just about any useful graphics function. A fast line draw for 320×400 256-color mode would be simple (although not as fast as would be possible in mode 13H). Fast image copies could be implemented by copying one-quarter of the image to one plane, one-quarter to the next plane, and so on for all four planes, thereby eliminating the OUT per pixel that sequential processing requires. If you’re really into performance, you could store your images with all the bytes for plane 0 grouped together, followed by all the bytes for plane 1, and so on. That would allow a single REP MOVBS instruction to copy all the bytes for a given plane, with just four REP MOVBS instructions copying the whole image. In a number of cases, in fact, 320×400 256-color mode can actually be much faster than mode 13H, because the VGA’s hardware can be used to draw four or even eight pixels with a single access; I’ll return to the topic of high-performance programming in 256-color modes other than mode 13H ("non-chain 4" modes) in Chapter 47.

It’s all a bit complicated, but as I say, you should be able to design an adequately fast—and often very fast—version for 320×400 mode of whatever graphics function you need. If you’re not all that concerned with speed, WritePixel and ReadPixel should meet your needs.

Two 256-Color Pages

Listing 31.2 demonstrates the two pages of 320×400 256-color mode by drawing slanting color bars in page 0, then drawing color bars slanting the other way in page 1 and flipping to page 1 on the next key press. (Note that page 1 is accessed starting at offset 8000H in display memory, and is—unsurprisingly—displayed by setting the start address to 8000H.) Finally, Listing 31.2 draws vertical color bars in page 0 and flips back to page 0 when another key is pressed.

The color bar routines don’t use the WritePixel subroutine from Listing 31.1; they go straight to display memory instead for improved speed. As I mentioned above, better speed yet could be achieved by a color-bar algorithm that draws all the pixels in plane 0, then all the pixels in plane 1, and so on, thereby avoiding the overhead of constantly reprogramming the Map Mask register.

LISTING 31.2  L31-2.ASM

; Program to demonstrate the two pages available in 320×400
; 256-color modes on a VGA. Draws diagonal color bars in all
; 256 colors in page 0, then does the same in page 1 (but with
; the bars tilted the other way), and finally draws vertical
; color bars in page 0.
;
VGA_SEGMENT equ 0a000h
SC_INDEX equ 3c4h ;Sequence Controller Index register
GC_INDEX equ 3cch ;Graphics Controller Index register
CRTC_INDEX equ 3d4h ;CRT Controller Index register
MAP_MASK equ 2 ;Map Mask register index in SC
MEMORY_MODE equ 4 ;Memory Mode register index in SC
MAX_SCAN_LINE equ 9 ;Maximum Scan Line reg index in CRTC
START_ADDRESS_HIGH equ 0ch ;Start Address High reg index in CRTC
UNDERLINE equ 14h ;Underline Location reg index in CRTC
MODE_CONTROL equ 17h ;Mode Control register index in CRTC
GRAPHICS_MODE equ 5 ;Graphics Mode register index in GC
MISCELLANEOUS equ 6 ;Miscellaneous register index in GC
SCREEN_WIDTH equ 320 ;# of pixels across screen
SCREEN_HEIGHT equ 400 ;# of scan lines on screen
WORD_OUTS_OK equ 1 ;set to 0 to assemble for
; computers that can't handle
; word cuts to indexed VGA registers

stack segment para stack 'STACK'
db 512 dup (?)
stack ends
;
; Macro to output a word value to a port.
;
OUT_WORD macro
if WORD_OUTS_OK
  out dx,ax
else
  out dx,al
  inc dx
  xchg ah,al
  out dx,al
  dec dx
  xchg ah,al
endif
endm
;
; Macro to output a constant value to an indexed VGA register.
;
CONSTANT_TO_INDEXED_REGISTER macro ADDRESS, INDEX, VALUE
  mov dx,ADDRESS
  mov ax,(VALUE shl 8) + INDEX
  OUT_WORD
endm
;
Code segment
assume cs:Code
Start proc near
;
; Set 320x400 256-color mode.
;
  call Set320By400Mode
;
; We're in 320x400 256-color mode, with page 0 displayed.
; Let's fill page 0 with color bars slanting down and to the right.
;
  sub di,di ;page 0 starts at address 0

Higher 256-Color Resolution on the VGA  601
mov bl,1
    ; make color bars slant down and
call ColorBarsUp
    ; to the right

; Wait for a key and flip to page 1 when one is pressed.
call GetNextKey
CONSTANT_TO_INDEXED_REGISTER CRTC_INDEX,START_ADDRESS_HIGH,80h
    ; set the Start Address High register
to 80h, for a start address of 8000h
; Draw vertical bars in page 0 while page 1 is displayed.
sub di,di
sub bl,bl
    ; page 0 starts at address 0
    ; make color bars vertical
call ColorBarsUp
    ; draw the color bars

; Wait for another key and flip back to page 0 when one is pressed.
call GetNextKey
CONSTANT_TO_INDEXED_REGISTER CRTC_INDEX,START_ADDRESS_HIGH,00h
    ; set the Start Address High register
to 00h, for a start address of 0000h
; Wait for yet another key and return to text mode and end when
; one is pressed.
call GetNextKey
mov ax,0003h
int 10h
    ; text mode
mov ah,4ch
int 21h
    ; done

Set320By400Mode proc near

mov ax,0013h ; AH = 0 means mode set, AL = 13h selects
to 13h selects
int 10h
    ; 256-color graphics mode
int 10h
    ; BIOS video interrupt

; Change CPU addressing of video memory to linear (not odd/even,
; chain, or chain 4), to allow us to access all 256K of display

602 Chapter 31
; memory. When this is done, VGA memory will look just like memory
; in modes 10h and 12h, except that each byte of display memory will
; control one 256-color pixel, with 4 adjacent pixels at any given
; address, one pixel per plane.
;
; mov dx,SC_INDEX
; mov al,MEMORY_MODE
; out dx,al
; inc dx
; in al,dx
; and al, not 08h ; turn off chain 4
; or al, 04h ; turn off odd/even
; out dx,al
; mov dx,SC_INDEX
; mov al,GRAPHICS_MODE
; out dx,al
; inc dx
; in al,dx
; and al, not 10h ; turn off odd/even
; out dx,al
; dec dx
; mov al,MISCELLANEOUS
; out dx,al
; inc dx
; in al,dx
; and al, not 02h ; turn off chain
; out dx,al
;
; : Now clear the whole screen, since the mode 13h mode set only
; : cleared 64K out of the 256K of display memory. Do this before
; : we switch the CRTC out of mode 13h, so we don't see garbage
; : on the screen when we make the switch.
; :
; CONSTANT_TO_INDEXED_REGISTER SC_INDEX,MAP_MASK,0fh
; : enable writes to all planes, so
; : we can clear 4 pixels at a time

; mov ax,VGA_SEGMENT
; mov es,ax
; sub di, di
; mov ax,di
; mov cx,8000h ; # of words in 64K
; cld
; rep stosw ; clear all of display memory
;
; : Tweak the mode to 320x400 256-color mode by not scanning each
; : line twice.
;
; mov dx,CRTC_INDEX
; mov al,MAX_SCAN_LINE
; out dx,al
; inc dx
; in al,dx
; and al, not 1fh ; set maximum scan line - 0
; out dx,al
; dec dx
;
; : Change CRTC scanning from doubleword mode to byte mode, allowing
; : the CRTC to scan more than 64K of video data.
;
; mov al,UNDERLINE
; out dx,al
inc dx
inc al,dx
and al,not 40h ; turn off doubleword
out dx,al
dec dx
mov al,MODE_CONTROL
out dx,al
inc dx
inc al,dx
or al,40h ; turn on the byte mode bit, so memory is
; scanned for video data in a purely
; linear way, just as in modes 10h and 12h
out dx,al
ret
Set320By400Mode endp

; Draws a full screen of slanting color bars in the specified page.
;
; Input:
; DI = page start address
; BL = 1 to make the bars slant down and to the right, -1 to
; make them slant down and to the left, 0 to make
; them vertical.
;
ColorBarsUp proc near
mov ax,VGA-SEGMENT
mov es,ax
sub bh,bh
mov si .SCREEN-HEIGHT
mov dx,SC-INDEX
mov a1 .MAP-MASK
out dx,al ; point the SC Index reg to the Map Mask reg
inc dx ; point DX to the SC Data register
mov cx,SCREEN_WIDTH/4
RowLoop:
; 4 pixels at each address, so
; each 320-pixel row is 80 bytes wide
; in each plane
; save the row-start color
push bx
ColumnLoop:
MAP_SELECT - 1
;do all 4 pixels at this address with
; in-line code
rept 4
mov a1,MAP_SELECT
out dx,al ; select planes 0, 1, 2, and 3 in turn
mov es:[di],bh ; write this plane’s pixel
inc bh
MAP_SELECT - MAP_SELECT shl 1
endm
inc di ; point to the address containing the next
; 4 pixels
loop ColumnLoop ; do any remaining pixels on this line
pop bx ; get back the row-start color
add bh,bl ; select next row-start color (controls
; slanting of color bars)
dec si ; count down lines on the screen
jnz RowLoop
ret
ColorBarsUp endp

604 Chapter 31
: Waits for the next key and returns it in AX.
: GetNextKey proc near
WaitKey:
    mov ah,1
    int 16h
    jz WaitKey ; wait for a key to become available
    sub ah,ah
    int 16h ; read the key
    ret
GetNextKey endp
;
Code ends
;
end Start

When you run Listing 31.2, note the extremely smooth edges and fine gradations of color, especially in the screens with slanting color bars. The displays produced by Listing 31.2 make it clear that 320×400 256-color mode can produce effects that are simply not possible in any 16-color mode.

**Something to Think About**

You can, if you wish, use the display memory organization of 320×400 mode in 320×200 mode by modifying Set320×400Mode to leave the maximum scan line setting at 1 in the mode set. (The version of Set320×400Mode in Listings 31.1 and 31.2 forces the maximum scan line to 0, doubling the effective resolution of the screen.) Why would you want to do that? For one thing, you could then choose from not two but four 320×200 256-color display pages, starting at offsets 0, 4000H, 8000H, and 0C000H in display memory. For another, having only half as many pixels per screen can as much as double drawing speeds; that’s one reason that many games run at 320×200, and even then often limit the active display drawing area to only a portion of the screen.