Chapter 34

Changing Colors without Writing Pixels

WHOOPS!

Our printer failed to strip in the art for Figure 34.1. The figure, however, is not essential to your understanding of this chapter. It's actually a screen shot of the output produced by Listing 34-5, a Mode X screen with a large number of small animated images zipping around. You can see what the figure should have been (and see it in color, and see it move, even!) by executing L34-5.EXE, which you will find in the listing archive subdirectory for Chapter 34 once you install the companion diskette.

Sorry for the omission.

--Jeff Duntemann, Editor

Chapter 34

Special Effects through Realtime Manipulation of DAC Colors

Sometimes, strange as it may seem, the harder you try, the less you accomplish. Brute force is fine when it suffices, but it does not always suffice, and when it does not, finesse and alternative approaches are called for. Such is the case with rapidly cycling through colors by repeatedly loading the VGA's Digital to Analog Converter (DAC). No matter how much you optimize your code, you just can't reliably load the whole DAC cleanly in a single frame, so you had best find other ways to use the DAC to cycle colors. What's more, BIOS support for DAC loading is so inconsistent that it's unusable for color cycling; direct loading through the I/O ports is the only way to go. We'll see why next, as we explore color cycling, and then finish up this chapter and this section by cleaning up some odds and ends about VGA color.

There's a lot to be said about loading the DAC, so let's dive right in and see where the complications lie.

Color Cycling

As we've learned in past chapters, the VGA's DAC contains 256 storage locations, each holding one 18-bit value representing an RGB color triplet organized as 6 bits per primary color. Each and every pixel generated by the VGA is fed into the DAC as an 8-bit value (refer to Chapter 33 and to Chapter A on the companion CD-ROM to see how pixels become 8-bit values in non-256 color modes) and each 8-bit value is used to look up one of the 256 values stored in the DAC. The looked-up value is then converted to analog red, green, and blue signals and sent to the monitor to form one pixel.

That's straightforward enough, and we've produced some pretty impressive color effects by loading the DAC once and then playing with the 8-bit path into the DAC. Now, however, we want to generate color effects by dynamically changing the values stored in the DAC in real time, a technique that I'll call *color cycling*. The potential of color cycling should be obvious: Smooth motion can easily be simulated by altering the colors in an appropriate pattern, and all sorts of changing color effects can be produced without altering a single bit of display memory.

For example, a sunset can be made to color and darken by altering the DAC locations containing the colors used to draw the sunset, or a river can be made to appear to flow by cycling through the colors used to draw the river. Another use for color cycling is in providing more realistic displays for applications like realtime 3-D games, where the VGA's 256 simultaneous colors can be made to seem like many more by changing the DAC settings from frame to frame to match the changing color demands of the rendered scene. Which leaves only one question: How do we load the DAC smoothly in realtime?

Actually, so far as I know, you can't. At least you can't load the *entire* DAC—all 256 locations—frame after frame without producing distressing on-screen effects on at least some computers. In non-256 color modes, it is indeed possible to load the DAC quickly enough to cycle all displayed colors (of which there are 16 or fewer), so color cycling could be used successfully to cycle all colors in such modes. On the other hand, color paging (which flips among a number of color sets stored within the DAC in all modes other than 256 color mode, as discussed in Chapter A on the companion CD-ROM) can be used in non-256 color modes to produce many of the same effects as color cycling and is considerably simpler and more reliable then color cycling, so color paging is generally superior to color cycling whenever it's available. In short, color cycling is really the method of choice for dynamic color effects only in 256-color mode—but, regrettably, color cycling is at its least reliable and capable in that mode, as we'll see next.

The Heart of the Problem

Here's the problem with loading the entire DAC repeatedly: The DAC contains 256 color storage locations, each loaded via either 3 or 4 **OUT** instructions (more on that next), so at least 768 **OUT**s are needed to load the entire DAC. That many **OUT**s take a considerable amount of time, all the more so because **OUT**s are painfully slow on 486s and Pentiums, and because the DAC is frequently on the ISA bus (although VLB and PCI are increasingly common), where wait states are inserted in fast computers. In an 8 MHz AT, 768 **OUT**s alone would take 288 microseconds, and the data loading and looping that are also required would take in the ballpark of 1,800 microseconds more, for a minimum of 2 milliseconds total.

As it happens, the DAC should only be loaded during vertical blanking; that is, the time between the end of displaying the bottom border and the start of displaying the top border, when no video information at all is being sent to the screen by the DAC. Otherwise, small dots of snow appear on the screen, and while an occasional dot of this sort wouldn't be a problem, the constant DAC loading required by color cycling would produce a veritable snowstorm on the screen. By the way, I do mean "border," not "frame buffer"; the overscan pixels pass through the DAC just like the pixels controlled by the frame buffer, so you can't even load the DAC while the border color is being displayed without getting snow.

The start of vertical blanking itself is not easy to find, but the leading edge of the vertical sync pulse is easy to detect via bit 3 of the Input Status 1 register at 3DAH; when bit 3 is 1, the vertical sync pulse is active. Conveniently, the vertical sync pulse starts partway through but not too far into vertical blanking, so it serves as a handy way to tell when it's safe to load the DAC without producing snow on the screen.

So we wait for the start of the vertical sync pulse, then begin to load the DAC. There's a catch, though. On many computers—Pentiums, 486s, and 386s sometimes, 286s most of the time, and 8088s all the time—there just isn't enough time between the start of the vertical sync pulse and the end of vertical blanking to load all 256 DAC locations. That's the crux of the problem with the DAC, and shortly we'll get to a tool that will let you explore for yourself the extent of the problem on computers in which you're interested. First, though, we must address *another* DAC loading problem: the BIOS.

Loading the DAC via the BIOS

The DAC can be loaded either directly or through subfunctions 10H (for a single DAC register) or 12H (for a block of DAC registers) of the BIOS video service interrupt 10H, function 10H, described in Chapter 33. For cycling the contents of the entire DAC, the block-load function (invoked by executing **INT** 10H with AH = 10H and AL = 12H to load a block of CX DAC locations, starting at location BX, from the block of RGB triplets—3 bytes per triplet—starting at ES:DX into the DAC) would be the better of the two, due to the considerably greater efficiency of calling the BIOS once rather than 256 times. At any rate, we'd like to use one or the other of the BIOS functions for color cycling, because we know that whenever possible, one should use a BIOS function in preference to accessing hardware directly, in the interests of avoiding compatibility problems. In the case of color cycling, however, it is emphatically *not* possible to use either of the BIOS functions, for they have problems. Serious problems.

The difficulty is this: IBM's BIOS specification describes exactly how the parameters passed to the BIOS control the loading of DAC locations, and all clone BIOSes meet that specification scrupulously, which is to say that if you invoke **INT** 10H, function 10H, subfunction 12H with a given set of parameters, you can be sure that you will end up with the same values loaded into the same DAC locations on all VGAs from all vendors. IBM's spec does *not*, however, describe whether vertical retrace should be waited for before loading the DAC, nor does it mention whether video should be left enabled while loading the DAC, leaving cloners to choose whatever approach they desire—and, alas, every VGA cloner seems to have selected a different approach.

I tested four clone VGAs from different manufacturers, some in a 20 MHz 386 machine and some in a 10 MHz 286 machine. Two of the four waited for vertical retrace before loading the DAC; two didn't. Two of the four blanked the display while loading the DAC, resulting in flickering bars across the screen. One showed speckled pixels spattered across the top of the screen while the DAC was being loaded. Also, not one was able to load all 256 DAC locations without showing *some* sort of garbage on the screen for at least one frame, but that's not the BIOS's fault; it's a problem endemic to the VGA.

These findings lead me inexorably to the conclusion that the BIOS should not be used to load the DAC dynamically. That is, if you're loading the DAC just once in preparation for a graphics session—sort of a DAC mode set—by all means load by way of the BIOS. No one will care that some garbage is displayed for a single frame; heck, I have boards that bounce and flicker and show garbage every time I do a mode set, and the amount of garbage produced by loading the DAC once is far less noticeable. If, however, you intend to load the DAC repeatedly for color cycling, avoid the BIOS DAC load functions like the plague. They will bring you only heartache.

As but one example of the unsuitability of the BIOS DAC-loading functions for color cycling, imagine that you want to cycle all 256 colors 70 times a second, which is once per frame. In order to accomplish that, you would normally wait for the start of the vertical sync signal (marking the end of the frame), then call the BIOS to load the DAC. On some boards—boards with BIOSes that don't wait for vertical sync before loading the DAC—that will work pretty well; you will, in fact, load the DAC once a frame. On other boards, however, it will work very poorly indeed; your program will wait for the start of vertical sync, and then the BIOS will wait for the start of the next vertical sync, with the result being that the DAC gets loaded only once every *two* frames. Sadly, there's no way, short of actually profiling the performance of BIOS DAC loads, for you to know which sort of BIOS is installed in a particular computer, so unless you can always control the brand of VGA your software will run on, you really can't afford to color cycle by calling the BIOS.

Which is not to say that loading the DAC directly is a picnic either, as we'll see next.

Loading the DAC Directly

So we must load the DAC directly in order to perform color cycling. The DAC is loaded directly by sending (with an **OUT** instruction) the number of the DAC location to be loaded to the DAC Write Index register at 3C8H and then performing three **OUT**s to write an RGB triplet to the DAC Data register at 3C9H. This approach must be repeated 256 times to load the entire DAC, requiring over a thousand **OUT**s in all.

There is another, somewhat faster approach, but one that has its risks. After an RGB triplet is written to the DAC Data register, the DAC Write Index register automatically increments to point to the next DAC location, and this repeats indefinitely as successive RGB triplets are written to the DAC. By taking advantage of this feature, the entire DAC can be loaded with just 769 **OUT**s: one **OUT** to the DAC Write Index register and 768 **OUT**s to the DAC Data register.

So what's the drawback? Well, imagine that as you're loading the DAC, an interrupt-driven TSR (such as a program switcher or multitasker) activates and writes to the DAC; you could end up with quite a mess on the screen, especially when your program resumes and continues writing to the DAC—but in all likelihood to the wrong locations. No problem, you say; just disable interrupts for the duration. Good idea—but it takes much longer to load the DAC than interrupts should be disabled for. If, on the other hand, you set the index for each DAC location separately, you can disable interrupts 256 times, once as each DAC location is loaded, without problems.

As I commented in the last chapter, I don't have any gruesome tale to relate that mandates taking the slower but safer road and setting the index for each DAC location separately while interrupts are disabled. I'm merely hypothesizing as to what ghastly mishaps *could* happen. However, it's been my experience that anything that can happen on the PC *does* happen eventually; there are just too dang many PCs out there for it to be otherwise. However, load the DAC any way you like; just don't blame me if you get a call from someone who's claims that your program sometimes turns their screen into something resembling month-old yogurt. It's not really your fault, of course—but try explaining that to *them!*

A Test Program for Color Cycling

Anyway, the choice of how to load the DAC is yours. Given that I'm not providing you with any hard-and-fast rules (mainly because there don't seem to be any), what you need is a tool so that you can experiment with various DAC-loading approaches for yourself, and that's exactly what you'll find in Listing 34.1.

Listing 34.1 draws a band of vertical lines, each one pixel wide, across the screen. The attribute of each vertical line is one greater than that of the preceding line, so there's a smooth gradient of attributes from left to right. Once everything is set up, the program starts cycling the colors stored in however many DAC locations are specified by the **CYCLE_SIZE** equate; as many as all 256 DAC locations can be cycled. (Actually, **CYCLE_SIZE**-1 locations are cycled, because location 0 is kept constant in order to keep the background and border colors from changing, but **CYCLE_SIZE** locations are *loaded*, and it's the number of locations we can load without problems that we're interested in.)

LISTING 34.1 L34-1.ASM

```
; Fills a band across the screen with vertical bars in all 256
; attributes, then cycles a portion of the palette until a key is
; pressed.
; Assemble with MASM or TASM

USE_BIOS               equ   1          ;set to 1 to use BIOS functions to access the
                                        ; DAC, 0 to read and write the DAC directly
GUARD_AGAINST_INTS     equ   1          ;1 to turn off interrupts and set write index
                                        ; before loading each DAC location, 0 to rely
                                        ; on the DAC auto-incrementing
WAIT_VSYNC             equ   1          ;set to 1 to wait for the leading edge of
                                        ; vertical sync before accessing the DAC, 0
                                        ; not to wait
NOT_8088               equ   0          ;set to 1 to use REP INSB and REP OUTSB when
                                        ; accessing the  DAC directly, 0 to use
                                        ; IN/STOSB and LODSB/OUT
CYCLE_SIZE             equ   256        ;# of DAC locations to cycle, 256 max
SCREEN_SEGMENT         equ   0a000h     ;mode 13h display memory segment
SCREEN_WIDTH_IN_BYTES  equ   320        ;# of bytes across the screen in mode 13h
INPUT_STATUS_1         equ   03dah      ;input status 1 register port
DAC_READ_INDEX         equ   03c7h      ;DAC Read Index register
DAC_WRITE_INDEX        equ   03c8h      ;DAC Write Index register
DAC_DATA               equ   03c9h      ;DAC Data register

if NOT_8088
      .286
endif ;NOT_8088

      .model     small
      .stack     100h
      .data
;Storage for all 256 DAC locations, organized as one three-byte
; (actually three 6-bit values; upper two bits of each byte aren't
; significant) RGB triplet per color.
PaletteTemp db     256*3 dup(?)
      .code
start:
      mov   ax,@data
      mov   ds,ax

;Select VGA's standard 256-color graphics mode, mode 13h.
      mov   ax,0013h                    ;AH = 0: set mode function,
      int   10h                         ; AL = 13h: mode # to set

;Read all 256 DAC locations into PaletteTemp (3 6-bit values, one
; each for red, green, and blue, per DAC location).

if WAIT_VSYNC
;Wait for the leading edge of the vertical sync pulse; this ensures
; that we read the DAC starting during the vertical non-display
; period.
      mov   dx,INPUT_STATUS_1
WaitNotVSync:                           ;wait to be out of vertical sync
      in    al,dx
      and   al,08h
      jnz   WaitNotVSync
WaitVSync:                              ;wait until vertical sync begins
      in    al,dx
      and   al,08h
```

```
     jz    WaitVSync
endif                                   ;WAIT_VSYNC

if USE_BIOS
     mov   ax,1017h                     ;AH = 10h: set DAC function,
                                        ; AL = 17h: read DAC block subfunction
     sub   bx,bx                        ;start with DAC location 0
     mov   cx,256                       ;read out all 256 locations
     mov   dx,seg PaletteTemp
     mov   es,dx
     mov   dx,offset PaletteTemp        ;point ES:DX to array in which
                                        ; the DAC values are to be stored
     int   10h                          ;read the DAC
else                                    ;!USE_BIOS
 if GUARD_AGAINST_INTS
     mov   cx,CYCLE_SIZE                ;# of DAC locations to load
     mov   di,seg PaletteTemp
     mov   es,di
     mov   di,offset PaletteTemp        ;dump the DAC into this array
     sub   ah,ah                        ;start with DAC location 0
DACStoreLoop:
     mov   dx,DAC_READ_INDEX
     mov   al,ah
     cli
     out   dx,al                        ;set the DAC location #
     mov   dx,DAC_DATA
     in    al,dx                        ;get the red component
     stosb
     in    al,dx                        ;get the green component
     stosb
     in    al,dx                        ;get the blue component
     stosb
     sti
     inc   ah
     loop  DACStoreLoop
 else ;!GUARD_AGAINST_INTS
     mov   dx,DAC_READ_INDEX
     sub   al,al
     out   dx,al                        ;set the initial DAC location to 0
     mov   di,seg PaletteTemp
     mov   es,di
     mov   di,offset PaletteTemp        ;dump the DAC into this array
     mov   dx,DAC_DATA
  if NOT_8088
     mov   cx,CYCLE_SIZE*3
     rep   insb                         ;read CYCLE_SIZE DAC locations at once
  else      ;!NOT_8088
     mov   cx,CYCLE_SIZE                ;# of DAC locations to load
DACStoreLoop:
     in    al,dx                        ;get the red component
     stosb
     in    al,dx                        ;get the green component
     stosb
     in    al,dx                        ;get the blue component
     stosb
     loop  DACStoreLoop
  endif                                 ;NOT_8088
 endif                                  ;GUARD_AGAINST_INTS
endif ;USE_BIOS
```

```
;Draw a series of 1-pixel-wide vertical bars across the screen in
; attributes 1 through 255.
      mov   ax,SCREEN_SEGMENT
      mov   es,ax
      mov   di,50*SCREEN_WIDTH_IN_BYTES       ;point ES:DI to the start
                                              ; of line 50 on the screen
      cld
      mov   dx,100                            ;draw 100 lines high
RowLoop:
      mov   al,1                              ;start each line with attr 1
      mov   cx,SCREEN_WIDTH_IN_BYTES          ;do a full line across
ColumnLoop:
      stosb                                   ;draw a pixel
      add   al,1                              ;increment the attribute
      adc   al,0                              ;if the attribute just turned
                                              ; over to 0, increment it to 1
                                              ; because we're not going to
                                              ; cycle DAC location 0, so
                                              ; attribute 0 won't change
      loop  ColumnLoop
      dec   dx
      jnz   RowLoop

;Cycle the specified range of DAC locations until a key is pressed.
CycleLoop:
;Rotate colors 1-255 one position in the PaletteTemp array;
; location 0 is always left unchanged so that the background
; and border don't change.
      push  word ptr PaletteTemp+(1*3)        ;set aside PaletteTemp
      push  word ptr PaletteTemp+(1*3)+2      ; setting for attr 1
      mov   cx,254
      mov   si,offset PaletteTemp+(2*3)
      mov   di,offset PaletteTemp+(1*3)
      mov   ax,ds
      mov   es,ax
      mov   cx,254*3/2
      rep   movsw                             ;rotate PaletteTemp settings
                                              ; for attrs 2 through 255 to
                                              ; attrs 1 through 254
      pop   bx                                ;get back original settings
      pop   ax                                ; for attribute 1 and move
      stosw                                   ; them to the PaletteTemp
      mov   es:[di],bl                        ; location for attribute 255

if WAIT_VSYNC
;Wait for the leading edge of the vertical sync pulse; this ensures
; that we reload the DAC starting during the vertical non-display
; period.
      mov   dx,INPUT_STATUS_1
WaitNotVSync2:                                ;wait to be out of vertical sync
      in    al,dx
      and   al,08h
      jnz   WaitNotVSync2
WaitVSync2:                                   ;wait until vertical sync begins
      in    al,dx
      and   al,08h
      jz    WaitVSync2
endif ;WAIT_VSYNC

if USE_BIOS
;Set the new, rotated palette.
```

```
      mov   ax,1012h                  ;AH = 10h: set DAC function,
                                      ; AL = 12h: set DAC block subfunction
      sub   bx,bx                     ;start with DAC location 0
      mov   cx,CYCLE_SIZE             ;# of DAC locations to set
      mov   dx,seg PaletteTemp
      mov   es,dx
      mov   dx,offset PaletteTemp  ;point ES:DX to array from which
                                      ; to load the DAC
      int   10h                       ;load the DAC
else  ;!USE_BIOS
 if GUARD_AGAINST_INTS
      mov   cx,CYCLE_SIZE             ;# of DAC locations to load
      mov   si,offset PaletteTemp     ;load the DAC from this array
      sub   ah,ah                     ;start with DAC location 0
DACLoadLoop:
      mov   dx,DAC_WRITE_INDEX
      mov   al,ah
      cli
      out   dx,al                     ;set the DAC location #
      mov   dx,DAC_DATA
      lodsb
      out   dx,al                     ;set the red component
      lodsb
      out   dx,al                     ;set the green component
      lodsb
      out   dx,al                     ;set the blue component
      sti
      inc   ah
      loop  DACLoadLoop
 else ;!GUARD_AGAINST_INTS
      mov   dx,DAC_WRITE_INDEX
      sub   al,al
      out   dx,al                     ;set the initial DAC location to 0
      mov   si,offset PaletteTemp     ;load the DAC from this array
      mov   dx,DAC_DATA
  if NOT_8088
      mov   cx,CYCLE_SIZE*3
      rep   outsb                     ;load CYCLE_SIZE DAC locations at once
  else      ;!NOT_8088
      mov   cx,CYCLE_SIZE             ;# of DAC locations to load
DACLoadLoop:
      lodsb
      out   dx,al                     ;set the red component
      lodsb
      out   dx,al                     ;set the green component
      lodsb
      out   dx,al                     ;set the blue component
      loop  DACLoadLoop
  endif     ;NOT_8088
 endif      ;GUARD_AGAINST_INTS
endif ;USE_BIOS

;See if a key has been pressed.
      mov   ah,0bh                    ;DOS check standard input status fn
      int   21h
      and   al,al                     ;is a key pending?
      jz    CycleLoop                 ;no, cycle some more

;Clear the keypress.
      mov   ah,1                      ;DOS keyboard input fn
      int   21h
```

```
;Restore text mode and done.
      mov    ax,0003h                         ;AH = 0: set mode function,
      int    10h                              ; AL = 03h: mode # to set
      mov    ah,4ch                           ;DOS terminate process fn
      int    21h

      end    start
```

The big question is, How does Listing 34.1 cycle colors? Via the BIOS or directly? With interrupts enabled or disabled? *Et cetera*?

However you like, actually. Four equates at the top of Listing 34.1 select the sort of color cycling performed; by changing these equates and **CYCLE_SIZE**, you can get a feel for how well various approaches to color cycling work with whatever combination of computer system and VGA you care to test.

The **USE_BIOS** equate is simple. Set **USE_BIOS** to 1 to load the DAC through the block-load-DAC BIOS function, or to 0 to load the DAC directly with **OUT**s.

If **USE_BIOS** is 1, the only other equate of interest is **WAIT_VSYNC**. If **WAIT_VSYNC** is 1, the program waits for the leading edge of vertical sync before loading the DAC; if **WAIT_VSYNC** is 0, the program doesn't wait before loading. The effect of setting or not setting **WAIT_VSYNC** depends on whether the BIOS of the VGA the program is running on waits for vertical sync before loading the DAC. You may end up with a double wait, causing color cycling to proceed at half speed, you may end up with no wait at all, causing cycling to occur far too rapidly (and almost certainly with hideous on-screen effects), or you may actually end up cycling at the proper one-cycle-per-frame rate.

If **USE_BIOS** is 0, **WAIT_VSYNC** still applies. However, you will always want to set **WAIT_VSYNC** to 1 when **USE_BIOS** is 0; otherwise, cycling will occur much too fast, and a good deal of continuous on-screen garbage is likely to make itself evident as the program loads the DAC non-stop.

If **USE_BIOS** is 0, **GUARD_AGAINST_INTS** determines whether the possibility of the DAC loading process being interrupted is guarded against by disabling interrupts and setting the write index once for every location loaded and whether the DAC's autoincrementing feature is relied upon or not.

If **GUARD_AGAINST_INTS** is 1, the following sequence is followed for the loading of each DAC location in turn: Interrupts are disabled, the DAC Write Index register is set appropriately, the RGB triplet for the location is written to the DAC Data register, and interrupts are enabled. This is the slow but safe approach described earlier.

Matters get still more interesting if **GUARD_AGAINST_INTS** is 0. In that case, if **NOT_8088** is 0, then an autoincrementing load is performed in a straightforward fashion; the DAC Write Index register is set to the index of the first location to load and the RGB triplet is sent to the DAC by way of three **LODSB/OUT DX,AL** pairs, with **LOOP** repeating the process for each of the locations in turn.

If, however, **NOT_8088** is 1, indicating that the processor is a 286 or better (perhaps **AT_LEAST_286** would have been a better name), then after the initial DAC Write Index value is set, all 768 DAC locations are loaded with a single **REP OUTSB**. This is clearly the fastest approach, but it runs the risk, albeit remote, that the loading sequence will be interrupted and the DAC registers will become garbled.

My own experience with Listing 34.1 indicates that it is sometimes possible to load all 256 locations cleanly but sometimes it is not; it all depends on the processor, the bus speed, the VGA, and the DAC, as well as whether autoincrementation and **REP OUTSB** are used. I'm not going to bother to report how many DAC locations I *could* successfully load with each of the various approaches, for the simple reason that I don't have enough data points to make reliable suggestions, and I don't want you acting on my comments and running into trouble down the pike. You now have a versatile tool with which to probe the limitations of various DAC-loading approaches; use it to perform your own tests on a sampling of the slowest hardware configurations you expect your programs to run on, then leave a generous safety margin.

One thing's for sure, though—you're not going to be able to cycle all 256 DAC locations cleanly once per frame on a reliable basis across the current generation of PCs. That's why I said at the outset that brute force isn't appropriate to the task of color cycling. That doesn't mean that color cycling can't be used, just that subtler approaches must be employed. Let's look at some of those alternatives.

Color Cycling Approaches that Work

First of all, I'd like to point out that when color cycling does work, it's a thing of beauty. Assemble Listing 34.1 so that it doesn't use the BIOS to load the DAC, doesn't guard against interrupts, and uses 286-specific instructions if your computer supports them. Then tinker with **CYCLE_SIZE** until the color cycling is perfectly clean on your computer. Color cycling looks stunningly smooth, doesn't it? And this is crude color cycling, working with the default color set; switch over to a color set that gradually works its way through various hues and saturations, and you could get something that looks for all the world like true-color animation (albeit working with a small subset of the full spectrum at any one time).

Given that, how can we take advantage of color cycling within the limitations of loading the DAC? The simplest approach, and my personal favorite, is that of cycling a portion of the DAC while using the rest of the DAC locations for other, non-cycling purposes. For example, you might allocate 32 DAC locations to the aforementioned sunset, reserve 160 additional locations for use in drawing a static mountain scene, and employ the remaining 64 locations to draw images of planes, cars, and the like in the foreground. The 32 sunset colors could be cycled cleanly, and the other 224 colors would remain the same throughout the program, or would change only occasionally.

That suggests a second possibility: If you have several different color sets to be cycled, interleave the loading so that only one color set is cycled per frame. Suppose you are animating a night scene, with stars twinkling in the background, meteors streaking across the sky, and a spaceship moving across the screen with its jets flaring. One way to produce most of the necessary effects with little effort would be to draw the stars in several attributes and then cycle the colors for those attributes, draw the meteor paths in successive attributes, one for each pixel, and then cycle the colors for *those* attributes, and do much the same for the jets. The only remaining task would be to animate the spaceship across the screen, which is not a particularly difficult task.

The key to getting all the color cycling to work in the above example, however, would be to assign each color cycling task a different part of the DAC, with each part cycled independently as needed. If, as is likely, the total number of DAC locations cycled proved to be too great to manage in one frame, you could simply cycle the colors of the stars after one frame, the colors of the meteors after the next, and the colors of the jets after yet another frame, then back around to cycling the colors of the stars. By splitting up the DAC in this manner and interleaving the cycling tasks, you can perform a great deal of seemingly complex color animation without loading very much of the DAC during any one frame.

Yet another and somewhat odder workaround is that of using only 128 DAC locations and page flipping. (Page flipping in 256-color modes involves using the VGA's undocumented 256-color modes; see Chapters 31, 43, and 47 for details.) In this mode of operation, you'd first display page 0, which is drawn entirely with colors 0-127. Then you'd draw page 1 to look just like page 0, except that colors 128-255 are used instead. You'd load DAC locations 128-255 with the next cycle settings for the 128 colors you're using, then you'd switch to display the second page with the new colors. Then you could modify page 0 as needed, drawing in colors 0-127, load DAC locations 0-127 with the next color cycle settings, and flip back to page 0.

The idea is that you modify only those DAC locations that are not used to display any pixels on the current screen. The advantage of this is *not*, as you might think, that you don't generate garbage on the screen when modifying undisplayed DAC locations; in fact, you do, for a spot of interference will show up if you set a DAC location, displayed or not, during display time. No, you still have to wait for vertical sync and load only during vertical blanking before loading the DAC when page flipping with 128 colors; the advantage is that since none of the DAC locations you're modifying is currently displayed, you can spread the loading out over two or more vertical blanking periods—however long it takes. If you did this without the 128-color page flipping, you might get odd on-screen effects as some of the colors changed after one frame, some after the next, and so on—or you might not; changing the entire DAC in chunks over several frames is another possibility worth considering.

Yet another approach to color cycling is that of loading a bit of the DAC during each horizontal blanking period. Combine that with counting scan lines, and you could vastly expand the number of simultaneous on-screen colors by cycling colors *as a frame is displayed,* so that the color set changes from scan line to scan line down the screen.

The possibilities are endless. However, were I to be writing 256-color software that used color cycling, I'd find out how many colors could be cycled after the start of vertical sync on the slowest computer I expected the software to run on, I'd lop off at least 10 percent for a safety margin, and I'd structure my program so that no color cycling set exceeded that size, interleaving several color cycling sets if necessary.

That's what *I'd* do. Don't let yourself be held back by my limited imagination, though! Color cycling may be the most complicated of all the color control techniques, but it's also the most powerful.

Odds and Ends

In my experience, when relying on the autoincrementing feature while loading the DAC, the Write Index register wraps back from 255 to 0, and likewise when you load a block of registers through the BIOS. So far as I know, this is a characteristic of the hardware, and should be consistent; also, Richard Wilton documents this behavior for the BIOS in the VGA bible, *Programmer's Guide to PC Video Systems, Second Edition* (Microsoft Press), so you should be able to count on it. Not that I see that DAC index wrapping is especially useful, but it never hurts to understand exactly how your resources behave, and I never know when one of you might come up with a serviceable application for any particular quirk.

The DAC Mask

There's one register in the DAC that I haven't mentioned yet, the DAC Mask register at 03C6H. The operation of this register is simple but powerful; it can mask off any or all of the 8 bits of pixel information coming into the DAC from the VGA. Whenever a bit of the DAC Mask register is 1, the corresponding bit of pixel information is passed along to the DAC to be used in looking up the RGB triplet to be sent to the screen. Whenever a bit of the DAC Mask register is 0, the corresponding pixel bit is ignored, and a 0 is used for that bit position in all look-ups of RGB triplets. At the extreme, a DAC Mask setting of 0 causes all 8 bits of pixel information to be ignored, so DAC location 0 is looked up for every pixel, and the entire screen displays the color stored in DAC location 0. This makes setting the DAC Mask register to 0 a quick and easy way to blank the screen.

Reading the DAC

The DAC can be read directly, via the DAC Read Index register at 3C7H and the DAC Data register at 3C9H, in much the same way as it can be written directly by way of the DAC Write Index register—complete with autoincrementing the DAC Read Index register after every three reads. Everything I've said about writing to the DAC applies to reading from the DAC. In fact, reading from the DAC can even cause snow, just as loading the DAC does, so it should ideally be performed during vertical blanking.

The DAC can also be read by way of the BIOS in either of two ways. **INT** 10H, function 10H (AH=10H), subfunction 15H (AL=15H) reads out a single DAC location, specified by BX; this function returns the RGB triplet stored in the specified location with the red component in the lower 6 bits of DH, the green component in the lower 6 bits of CH, and the blue component in the lower 6 bits of CL.

INT 10H, function 10H (AH=10H), subfunction 17H (AL=17H) reads out a block of DAC locations of length CX, starting with the location specified by BX. ES:DX must point to the buffer in which the RGB values from the specified block of DAC locations are to be stored. The form of this buffer (RGB, RGB, RGB ..., with three bytes per RGB triple) is exactly the same as that of the buffer used when calling the BIOS to load a block of registers.

Listing 34.1 illustrates reading the DAC both through the BIOS block-read function and directly, with the direct-read code capable of conditionally assembling to either guard against interrupts or not and to use **REP INSB** or not. As you can see, reading the DAC settings is very much symmetric with setting the DAC.

Cycling Down

And so, at long last, we come to the end of our discussion of color control on the VGA. If it has been more complex than anyone might have imagined, it has also been most rewarding. There's as much obscure but very real potential in color control as there is anywhere on the VGA, which is to say that there's a very great deal of potential indeed. Put color cycling or color paging together with the page flipping and image drawing techniques explored elsewhere in this book, and you'll leave the audience gasping and wondering "How the heck did they *do* that?"