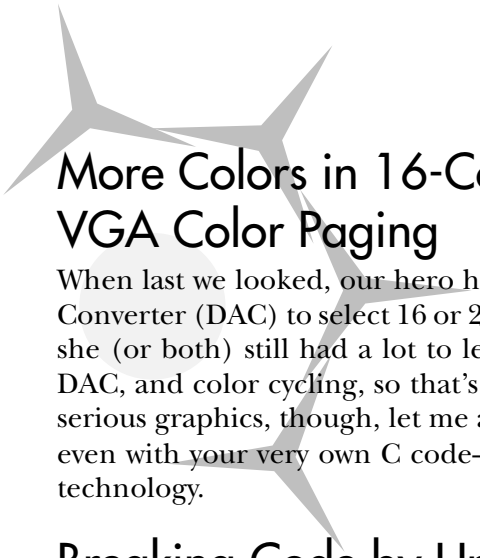
Chapter A

Paging Mr. VGA...

Chapter

A



More Colors in 16-Color Mode through VGA Color Paging

When last we looked, our hero had learned how to set the VGA's Digital to Analog Converter (DAC) to select 16 or 256 colors from a set of 256K colors. However, he or she (or both) still had a lot to learn about color paging, loading and reading the DAC, and color cycling, so that's where we're headed next. Before we get into any serious graphics, though, let me address a potential problem you may encounter—even with your very own C code—as you upgrade your programming tools to new technology.

Breaking Code by Upgrading Compilers

In its day, Microsoft C 6.0 was a solid compiler, but it broke much of the C graphics code I had presented in my various writings up to its release. You see, in order to set a single pixel in EGA/VGA memory, it's necessary to perform a read to latch display memory followed by a write to draw the pixel. (If you don't understand why this is the case, I suggest that you refer to Chapters 40–45 and ahead to Chapters F and G, but for now, just take my word for it.) Ideally we'd like to perform the read and write in close succession, with a single instruction if possible. In assembly language, that's no problem; just **XCHG**, **AND**, **XOR**, or **OR** an appropriate value with memory. In the code in question, the value actually written to memory is irrelevant, because of

the way set/reset and the bit mask are set up; all that's needed is a read followed by a write with any value at all.

Matters are not so simple in C. Long ago, I tried to use code like this

```
*ScreenPtr |= 0;
```

to perform a read/write operation to display memory. Unfortunately, the optimizer in Microsoft C 5.0 (not 6.0) considered this to be a null operation, because any value ORed with 0 remains unchanged (when the destination is system memory, that is; as described above, ORing with 0 is a useful operation when performed to VGA memory). Having concluded that it had encountered what amounted to a null operation, MSC 5.0 then declined to generate any code at all.

That was easy enough to fix. I just substituted

```
*ScreenPtr |= 0xFF;
```

which is definitely not a null operation (remember, for our purposes the value ORed with display memory is irrelevant, so 0xFF served just as well as 0). MSC 5.0 obligingly assembled the desired OR with memory, and there matters stood until the arrival of MSC 6.0.

MSC 6.0 doesn't think ***ScreenPtr |= 0xFF;** is a null operation—but it doesn't think it's an OR operation either. ORing a value with 0xFF invariably produces 0xFF as the result, so MSC 6.0 treated that code as if it were

```
*ScreenPtr = 0xFF;
```

and assembled a **MOV** instruction, not an **OR** instruction. That eliminated the read we needed before writing to display memory, and that's what broke so much graphics code.

The solution is simple: Instead of

```
*ScreenPtr |= 0xFF;
```

use

```
*ScreenPtr |= 0xFE;
```

and you should be all set.

There's an important point to all this beyond simply getting read-modify-write operations to work in C. Many people believe that it's possible to write low-level graphics code in C that's as efficient as code written in assembly language, and more portable, if you know exactly what code the compiler generates. This example shows why that's a fool's game; code that depends on the compiler's code generation isn't necessarily even portable from one release of the compiler to the next, let alone between compilers. Besides, the graphics adapters you're likely to encounter under MS-DOS are generally found only in x86-based computers, so it's

hard to imagine what portability benefits there might be. On top of all that, I'm still waiting to encounter C graphics code that matches good assembly language code in the performance department.

All in all, my advice is to stick to assembly language for your key graphics code. And with that out of the way, let's move on to color paging.

Color Paging

As you'll recall from Chapter 33, the VGA's DAC contains 256 storage locations, each holding one 18-bit color organized as one 6-bit red component, one 6-bit green component, and one 6-bit blue component. In 256-color mode, each 8-bit pixel value in memory selects one of the 256 DAC locations, the value in that location is looked up and sent to the screen as the pixel's color, and that's that. In 16-color modes, however, each pixel is represented in display memory by a 4-bit value, so each pixel can only look up one of 16 DAC locations to be sent to the screen. What about those other 240 DAC locations? Are they useless in 16-color modes?

Not at all. Normally (as configured by the BIOS at mode set time in 16-color modes), the first 64 locations in the DAC are set to colors that are equivalent to the 64 colors an EGA can display. (The other 192 DAC locations may or may not be set to any particular values, so be sure to load them before relying on their contents.) At the same time, the EGA-compatible palette RAM is set to the same values an EGA is normally set to. This is dandy if you're writing code for an EGA, because when you tweak the palette RAM on a VGA you'll get exactly the results you'd expect on an EGA. If you're writing VGA-specific code, however, you're shortchanging yourself.

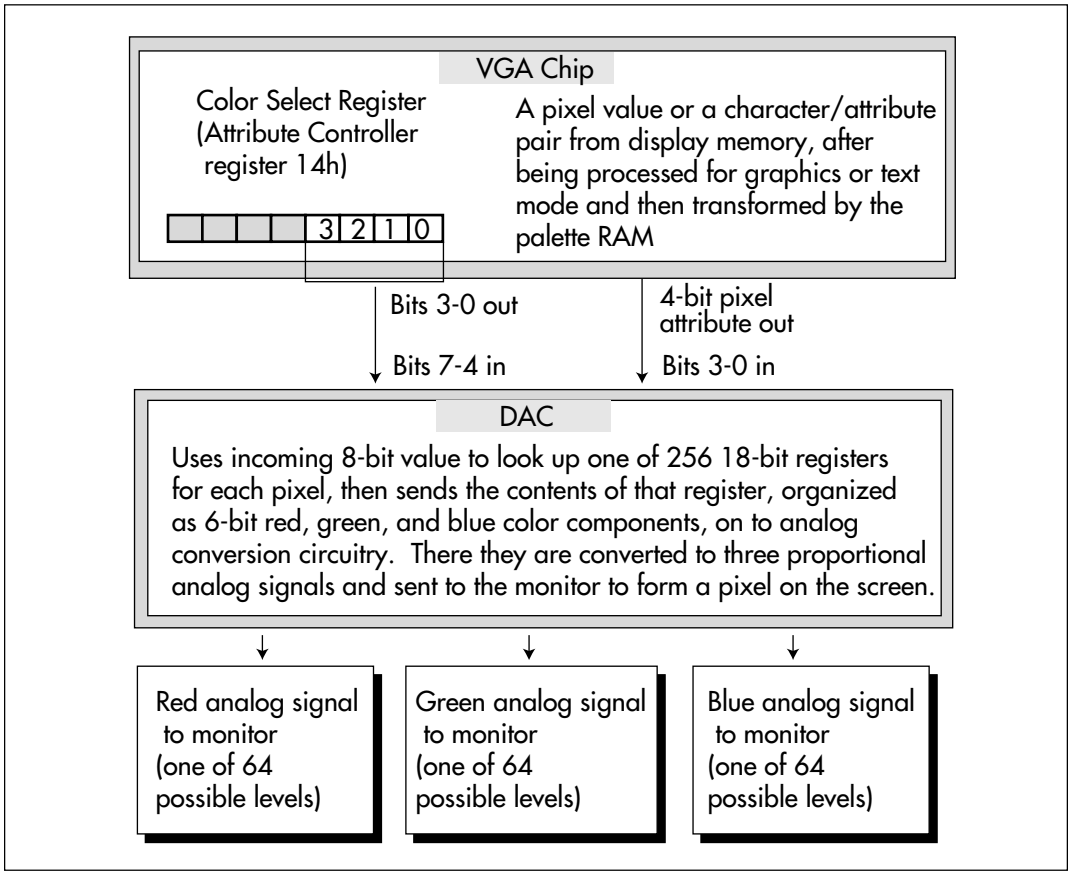
The ideal 16-color arrangement on a VGA is as follows: Load palette RAM register 0 with the value 0, palette RAM register 1 with the value 1, and so on up to palette RAM register 15, which should be set to 15. (I described how to load the palette RAM in Chapter 33, and in a little while we'll see working code that loads the palette.) The object here is to cause the palette RAM to pass pixel values through unchanged, so we can ignore it; the DAC will do all the color work.

Now load DAC locations 0 through 15 with any 16 colors you'd care to display. A pixel value of 0 in memory will cause the color in DAC location 0 to be displayed, a pixel value of 1 in memory will select DAC location 1, and so on. You can change the values stored in the DAC whenever you want to work with a different set of colors. (Loading the DAC was discussed in Chapter 33, and will be revisited in detail in Chapter 34.)

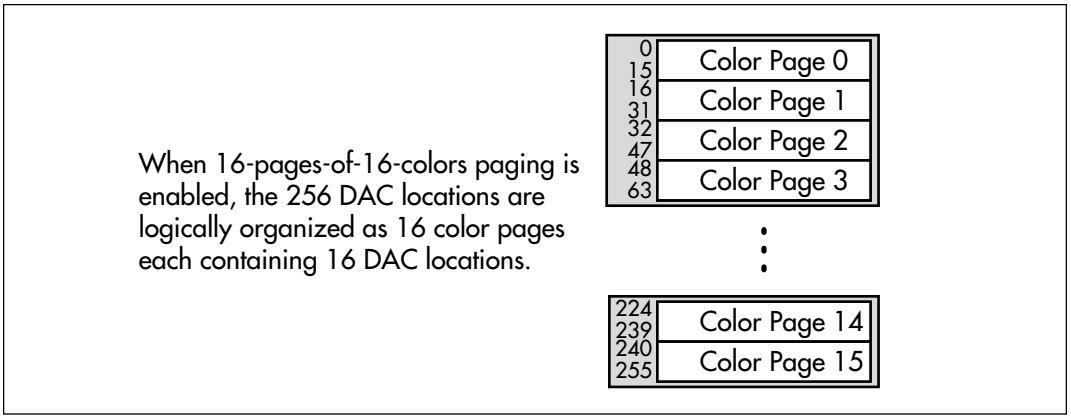
Things get more exciting with the addition of color paging. If you set bit 7 of the Attribute Controller Mode register (Attribute Controller register 10H) to 1, bits 3 through 0 of the Color Select register (Attribute Controller register 14H) become bits 7 through 4 of the 8-bit values used to address locations within the DAC. The Color Select register operates in conjunction with the pixel data from display memory,

which provides DAC address bits 3 through 0, as shown in Figure A.1. I'll call this mode of operation "16-pages-of-16-colors" paging. When bit 7 of the AC Mode register is 0 (the EGA-compatible default), "4-pages-of-64-colors" paging is selected, as described in Chapter 33.

The Color Select register gives us the ability to perform color paging; that is, to flip instantly between color sets and thereby provide a new color interpretation for every pixel on the screen without changing the contents of display memory. If you view the DAC as consisting of 16 pages each containing 16 colors, as shown in Figure A.2, then the Color Select register selects one of those pages and the pixel data selects one of 16 colors from within the currently selected page. (Remember that the palette RAM is set to a pass-through state, so pixel values of 0 through 15 come through the palette RAM to the DAC unaltered.) Basically, the Color Select register gives you



Color selection with 16-pages-of-16-colors paging.
Figure A.1



16-pages-of-16-colors paging.
Figure A.2

instantaneous access to any one of 16 completely independent pages of 16 colors each. (By the way, color paging isn't available in 256-color mode. The reason should be obvious; there are only enough DAC locations for one set of 256 colors.)

Big deal, you say; why not just reprogram the first 16 DAC locations if you want different colors? First, it's considerably faster to program the Color Select register with a few I/O operations than it is to reprogram 16 DAC registers, a process which takes 64 **OUT** instructions. Second, it's much easier to avoid flicker when using the Color Select register; all you have to do is wait for the vertical sync pulse and set one register. In fact, you don't even have to wait for the vertical sync pulse, because the colors will change instantly, with no glitching, starting at the location of the electron beam at the very instant you set the Color Select register. The only reason to wait for the sync pulse is to make sure that all the pixels in each frame are drawn with the same color set, in order to avoid having the top and bottom of the screen appear briefly in a mismatched state.

On the other hand, loading the DAC without flicker or glitching is no picnic. The loading has to take place during non-display time; otherwise for a short time part of the screen will be drawn with a mix of the old color set and the new color set, resulting in unintended and often highly undesirable on-screen effects. However, it can be difficult on slow computers to load a large block of DAC locations during a single vertical blanking period, so there may be no way to reload the DAC cleanly between one frame and the next (although this is more of a problem in 256-color mode, where it's sometimes necessary to reload all 256 DAC locations at once). Furthermore, some BIOSes glitch or bounce the screen when called to load the DAC, and some are faster than others.

We'll discuss issues of loading the DAC in Chapter 34. For now, what all this amounts to is that if you need to switch color sets frequently and color paging can do the job you need done, it's far superior to constantly reloading the DAC—and easier, too.

How to Perform Color Paging

Color paging can be performed either directly or through the BIOS. As usual, the BIOS route is the better choice if there's no good reason not to use it, but there is one possible drawback that we'll come across shortly. In any case, I'll describe both techniques.

Color paging directly (without the BIOS) is a two-step process: The first step is enabling color paging, and the second step is selecting the desired color page. Enabling color paging such that you have 16 pages of 16 colors is a simple matter of setting bit 7 of the AC Mode register to 1, and is accomplished as follows:

- Read the Input Status 1 register at 3BAH (in monochrome mode) or 3DAH (in color mode) to reset the AC Index/Data toggle.
- Write 30H to the AC Index register at 3C0H to address AC register 10H, the AC Mode register (the index value is 30H rather than 10H because bit 5 of the AC Index register should always be 1 except when setting the palette RAM; when bit 5 is 0, the screen blanks).
- Read the AC Mode register setting from the AC Data register at 3C1H.
- OR 80H with the value just read to set bit 7 (this selects 16-pages-of-16-colors operation rather than 4-pages-of-64-colors operation, the default).
- Write the result to the AC Mode register via the AC Data register at 3C0H. (Remember that the AC Data register is at 3C1H for reads, but is at 3C0H on every other **OUT**—alternating with the AC Index register—for writes, in order to provide EGA compatibility.)

At this point, color paging with 16 pages of 16 colors each is enabled.

Once color paging is enabled, you can select a color page as follows:

- Read the Input Status 1 register at 3XAH to reset the AC Index/Data toggle.
- Write 34H to the AC Index register at 3C0H to address AC register 14H, the Color Select register.
- Write the desired page number (0 through 15) to the Color Select register via the AC Data register at 3C0H.

Repeat this process whenever you want to switch to another color page.

Although it's not required, you'll generally want to wait for the leading edge of the vertical sync pulse before switching pages, as this provides the smoothest color transition, as described above. Take the vertical sync wait out of Listing A.1 (which I'll present a little later) and set ***USE_BIOS*** *to 0, and you'll see that the appearance of the program suffers considerably.*

Controlling color paging through the BIOS is a similar two-step process. The first step is to enable 16-pages-of-16-colors paging, and that's done by invoking interrupt 10H, the video interrupt, with AH = 10H, AL = 13H, BL = 0, and BH = 1. (If BH = 0, 4-pages-of-64-colors operation is selected.) The second step is to select the desired color page by invoking interrupt 10H with AH = 10H, AL = 13H, BL = 1, and BH = the number of the desired page, in the range 0 through 15.

Of those I've seen, at least one VGA BIOS waits for the leading edge of the vertical sync pulse before switching color pages, and at least one VGA BIOS does not. This poses a significant problem; if you wait for the leading edge of vertical sync before calling the BIOS to switch pages and then the BIOS waits for the leading edge of vertical sync before switching pages, you'll only switch pages once every two frames at most. On the other hand, if you don't wait for vertical sync and neither does the BIOS, you'll end up switching pages in the middle of frames and doing so much too often. You can determine which is the case in your color page-flipping programs by timing how long it takes the BIOS to do several color page switches at start-up, although that's certainly a nuisance. As usual, only with your own software that programs the hardware directly do you have complete control and full knowledge of what's going on.

At any rate, you can try it both ways, because Listing A.1 supports both BIOS and direct color paging. In Listing A.1, I assume that the BIOS does wait for the leading edge of the vertical sync pulse.

Obtaining the Color Paging State

If you want to check whether 16-pages-of-16-colors paging is enabled and which color page is currently selected, you can again do that either directly or through the BIOS. Obtaining the color paging state through the BIOS is ridiculously easy: Just invoke INT 10H with AH = 10H and AL = 1AH. On return, BL will contain the state of bit 7 of the AC Mode register (1 for 16-pages-of-16-colors paging, 0 for 4-pages-of-64-colors), and BH will contain the number of the currently selected color page.

To obtain the color paging state directly, do the following:

- Read the Input Status 1 register at 3XAH to reset the AC Index/Data toggle.
- Write 30H to the AC Index register at 3C0H to address the AC Mode register.
- Read the AC Mode register setting from the AC Data register at 3C1H. Bit 7 of this value controls the color paging state, as described above.
- Read the Input Status 1 register at 3XAH to reset the AC Index/Data toggle.
- Write 34H to the AC Index register at 3C0H to address the Color Select register.
- Read the Color Select register setting from the AC Data register at 3C1H.

An Example of Color Paging

Listing A.1 is an example of color paging. Listing A.1 first sets the display to 640×480 16-color graphics mode, then sets the palette RAM to a pass-through state and loads the upper 240 DAC locations with 240 colors, which are logically organized as 15 pages of 16 colors each. The palette RAM and the DAC must be set up *after* the mode set, because each mode set reprograms all of the palette RAM and at least part of the DAC. During EGA compatible mode sets, the first 64 DAC locations are programmed to match the EGA's set of colors, and during mode sets for mode 13H, 256-color mode, all 256 DAC locations are programmed.

Once the hardware is set up, Listing A.1 draws dozens of concentric circles, using code we developed in Chapters 34–37. I've used C rather than assembly language circle-drawing code (both versions are provided in Chapters B–E) because drawing speed isn't an issue here; Listing A.1 is written entirely in C, and while the drawing part of Listing A.1 looks slow, the color paging part does not. By its very nature, which involves working with a few control registers rather than a large bitmap, color manipulation tends to produce snappy results with little effort.

LISTING A.1 LA-1.C

```
/*
 * Illustrates color paging by color-animating a series of
 * concentric circles to produce the illusion of motion.
 * Runs on the VGA only, because color paging isn't available on
 * the EGA.
 */

#include <dos.h>

#define USE_BIOS  0  /* set to 1 to use BIOS functions to perform
                        color paging, 0 to program color paging
                        registers directly */



#define SCREEN_WIDTH_IN_BYTES     80          /* # of bytes across one scan
                                                 line in mode 12h */
#define SCREEN_SEGMENT            0xA000      /* mode 12h display memory seg */
#define GC_INDEX                  0x3CE       /* Graphics Controller index */
#define SET_RESET_INDEX           0           /* Set/Reset reg index in GC */
#define SET_RESET_ENABLE_INDEX    1           /* Set/Reset Enable reg index
                                                 in GC */
#define BIT_MASK_INDEX            8           /* Bit Mask reg index in GC */
#define INPUT_STATUS_1            0x3DA       /* Input Status 1 port */
#define AC_INDEX                  0x3C0       /* Attribute Controller index */
#define AC_DATA_W                 0x3C0       /* Attribute Controller data
                                                 register for writes */
#define AC_DATA_R                 0x3C1       /* Attribute Controller data
                                                 register for reads */

#define AC_MODE_INDEX             0x30        /* AC Mode reg index, with bit 6
                                                 set to avoid blanking screen */
#define AC_COLOR_SELECT_INDEX     0x34        /* Color Select reg index, with
                                                 bit 6 set to avoid blanking
                                                 screen */
```

```
void main();
void DrawDot(int X, int Y);
void DrawCircle(int X, int Y, int Radius, int Color);

/* Array used to load the DAC. Organized as 256 RGB triplets */
static unsigned char DACSettings[256*3];

/* Array used to load the palette RAM to a pass-through state.
   The 17th entry sets the border color to 0 */
static unsigned char PaletteRAMSettings[] = {
   0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0};

void main() {
   int Radius, Color, Page, Element, i;
   union REGS Regs;
   struct SREGS Sregs;
   unsigned char GreenComponent;

   /* Select VGA's hi-res 640x480 graphics mode, mode 12h */
   Regs.x.ax = 0x0012;
   int86(0x10, &Regs, &Regs);

   /* Draw concentric circles */
   for ( Radius = 10, Color = 1; Radius < 240; Radius += 2 ) {
      DrawCircle(640/2, 480/2, Radius, Color);
      if (++Color >= 16)
         Color = 1;                /* skip color 0 */
   }

   /* Load the upper 240 DAC locations (15 pages) with one-position
      rotations of a series of increasingly green colors. Because
      page 0 is being displayed, the screen remains unchanged while
      the other 15 color pages are being loaded. */

   /* First, fill DACSettings with the desired green settings
      (because it's a static array, all locations are initialized to
      zero, so we don't need to initialize the red or green color
      components, which we want to be zero). */
   GreenComponent = 8;
   for ( Page = 1; Page <= 15; Page++ ) {
      GreenComponent = Page * 4;
      for ( Element = 1; Element <= 15; Element++ ) {
         DACSettings[Page*16*3 + Element*3 + 1] = GreenComponent;
         if ( (GreenComponent += 4) >= 64 )
            GreenComponent = 4;
      }
   }

   /* Now call the BIOS to load the upper 240 DAC locations */
   Regs.h.ah = 0x10;
   Regs.h.al = 0x12;
   Regs.x.bx = 16;
   Regs.x.cx = 240;
   Regs.x.dx = (unsigned int)(DACSettings + 16*3);
   segread(&Sregs);
   Sregs.es = Sregs.ds;       /* point ES:DX to DACSettings */
   int86x(0x10, &Regs, &Regs, &Sregs);

   /* Put the palette RAM in a pass-through state and set the Overscan
      register (border color) to 0. We've saved this for last because
      it changes the colors being displayed. */
```

```
   Regs.h.ah = 0x10;
   Regs.h.al = 2;
   Regs.x.dx = (unsigned int)PaletteRAMSettings;
   segread(&Sregs);
   Sregs.es = Sregs.ds;       /* point ES:DX to PaletteRAMSettings */
   int86x(0x10, &Regs, &Regs, &Sregs);

   /* Enable 16-pages-of-16-colors paging */
#if USE_BIOS
   Regs.h.ah = 0x10;
   Regs.h.al = 0x13;
   Regs.h.bl = 0;
   Regs.h.bh = 1;
   int86(0x10, &Regs, &Regs);
#else
   inp(INPUT_STATUS_1);
   outp(AC_INDEX, AC_MODE_INDEX);
   outp(AC_DATA_W, inp(AC_DATA_R) | 0x80);
#endif /* USE_BIOS */

   /* We're ready to roll; the upper 15 pages are set up, the
      palette RAM is in a pass-through state, and 16-pages-of-16-
      colors paging is enabled. Now we'll loop through and display
      each of pages 15 through 1 and then back to 15 for one frame
      until a key is pressed. */
   for ( Page = 15, i = 0 ; i < 1000; i++ ) {

#if USE_BIOS
      /* Select the desired color page */
      Regs.h.ah = 0x10;
      Regs.h.al = 0x13;
      Regs.h.bl = 1;
      Regs.h.bh = Page;
      int86(0x10, &Regs, &Regs);
#else
      /* Wait for the leading edge of the vertical sync pulse; this
         ensures that we change color pages during vertical
         non-display time, and that the page flips are even spaced
         over time */
      while ( (inp(INPUT_STATUS_1) & 0x08) != 0 )
         ;                 /* wait for non-vertical sync time */
      while ( (inp(INPUT_STATUS_1) & 0x08) == 0 )
         ;                 /* wait for vertical sync time */
      inp(INPUT_STATUS_1);
      outp(AC_INDEX, AC_COLOR_SELECT_INDEX);
      outp(AC_DATA_W, Page);
#endif /* USE_BIOS */


      /* Cycle from page 15 down to page 1, and then back to page 15.
         Avoid page 0 entirely */
      if (--Page == 0)
         Page = 15;
   }

   /* Restore text mode and done */
   Regs.x.ax = 0x0003;
   int86(0x10, &Regs, &Regs);
}
```

```
/* Draws a pixel at screen coordinate (X,Y) */
void DrawDot(int X, int Y) {
   unsigned char far *ScreenPtr;

   /* Point to the byte the pixel is in */
#ifdef __TURBOC__
   ScreenPtr = MK_FP(SCREEN_SEGMENT,
      (Y * SCREEN_WIDTH_IN_BYTES) + (X / 8));
#else
   FP_SEG(ScreenPtr) = SCREEN_SEGMENT;
   FP_OFF(ScreenPtr) =(Y * SCREEN_WIDTH_IN_BYTES) + (X / 8);
#endif

   /* Set the bit mask within the byte for the pixel */
   outp(GC_INDEX + 1, 0x80 >> (X & 0x07));

   /* Draw the pixel. ORed to force read/write to load latches.
      Data written doesn't matter, because set/reset is enabled
      for all planes. Note: don't OR with 0; MSC optimizes that
      statement to no operation. */
   *ScreenPtr |= 0xFF;
}

/* Draws a circle of radius Radius in color Color centered at
 * screen coordinate (X,Y) */
void DrawCircle(int X, int Y, int Radius, int Color) {
   int MajorAxis, MinorAxis;
   unsigned long RadiusSqMinusMajorAxisSq;
   unsigned long MinorAxisSquaredThreshold;

   /* Set drawing color via set/reset */
   outpw(GC_INDEX, (0x0F << 8) | SET_RESET_ENABLE_INDEX);
                                /* enable set/reset for all planes */
   outpw(GC_INDEX, (Color << 8) | SET_RESET_INDEX);
                                /* set set/reset (drawing) color */
   outp(GC_INDEX, BIT_MASK_INDEX);
                                /* leave the GC Index reg pointing to
                                   the Bit Mask reg */

   /* Set up to draw the circle by setting the initial point to one
      end of the 1/8th of a circle arc we'll draw */
   MajorAxis = 0;
   MinorAxis = Radius;
   /* Set initial Radius**2 - MajorAxis**2 (MajorAxis is initially 0) */
   RadiusSqMinusMajorAxisSq = Radius * Radius;
   /* Set threshold for minor axis movement at (MinorAxis - 0.5)**2 */
   MinorAxisSquaredThreshold = MinorAxis * MinorAxis - MinorAxis;

   /* Draw all points along an arc of 1/8th of the circle, drawing
      all 8 symmetries at the same time */
   do {
      /* Draw all 8 symmetries of current point */
      DrawDot(X+MajorAxis, Y-MinorAxis);
      DrawDot(X-MajorAxis, Y-MinorAxis);
      DrawDot(X+MajorAxis, Y+MinorAxis);
      DrawDot(X-MajorAxis, Y+MinorAxis);
      DrawDot(X+MinorAxis, Y-MajorAxis);
      DrawDot(X-MinorAxis, Y-MajorAxis);
      DrawDot(X+MinorAxis, Y+MajorAxis);
      DrawDot(X-MinorAxis, Y+MajorAxis);
```

```
      /* Advance (Radius**2 - MajorAxis**2); if it equals or passes
         the MinorAxis**2 threshold, advance one pixel along the minor
         axis and set the next MinorAxis**2 threshold. */
      if ( (RadiusSqMinusMajorAxisSq -=
         MajorAxis + MajorAxis + 1) <= MinorAxisSquaredThreshold ) {
         MinorAxis--;
         MinorAxisSquaredThreshold -= MinorAxis + MinorAxis;
      }
      MajorAxis++;         /* advance one pixel along the major axis */
   } while ( MajorAxis <= MinorAxis );

   /* Reset the Bit Mask register to normal */
   outp(GC_INDEX + 1, 0xFF);

   /* Turn off set/reset enable */
   outpw(GC_INDEX, (0x00 << 8) | SET_RESET_ENABLE_INDEX);
}
```

The circles in Listing A.1 are drawn in sequential colors 1 through 15, cycling back from 15 to 1. Color 0 is reserved for the background and is the same in all 16 color pages; otherwise the background would change color as we changed color pages, destroying the subtle color cycling effect we're striving for. The background wouldn't be the only potential problem, either; if page 0 is involved, the border could also change, creating a still more bizarre effect. Remember, the DAC modifies pixels after they've been fully processed by the VGA, and so far as the DAC is concerned, pixel attribute 0 is attribute 0, irrespective of whether it came from the Overscan (border) register, a background pixel with value 0, or, indeed, by way of translation in the palette RAM. Consequently, you should take care when performing color paging not to modify the DAC locations that the Overscan register and background colors select, unless, of course, you intend to change the border and background colors.

This is a good time to point out that the DAC is *always* available and active. It's obvious that the DAC can be used to select color sets in 256- and 16-color modes, but the DAC processes whatever comes out of the VGA in any mode. You can use the DAC to transform pixel colors in text mode, or even in modes 4 and 6, the CGA-compatible 4- and 2-color graphics modes (although in order to do that you need to understand the format in which pixels comes out of the VGA and into the DAC in those modes; once again, setting the palette to a pass-through state makes life easier). In fact, you can also use color paging in any mode other than 256-color mode. The basic principle here is that the color-processing steps that occur farther down the pipeline toward the display can modify anything that comes earlier. As the last stage in the pipeline, the DAC can modify pixels in any mode at all, and as the lead-in to the DAC stage, color paging can modify pixels in all modes except 256-color mode.

Back to Listing A.1. Once the circles are drawn, Listing A.1 switches from the default color page, page 0, which we haven't changed, to page 15, and then starts flipping through the color pages in the order 14, 13, and so on down to 1 and then back round to 15 at the rate of one page per frame, or 60 times per second. What does this accomplish? Well, the color pages were carefully selected as one-position rotations

of increasingly bright green colors. That is, color page 1 translates a pixel value of 1 into dim green, a pixel value of 2 into next-to-dimmest green, and so on up to a pixel value of 15, which is as green a color as the VGA can produce. Color page 2 is a one-position rotation of color page 1; a pixel value of 1 produces next-to-dimmest green, a pixel value of 2 produces slightly brighter green, a pixel value of 14 produces brightest green, and a pixel value of 15 produces dimmest green. Each successive color page is a one-position rotation of the preceding page, except for color page 0, which isn't used because we need only 15 color pages to represent all possible rotations of a set of 15 colors.

Cycling through these color pages causes the circles on the screen to seem to pulse outward. The net effect is one of smooth motion, but not one pixel in display memory is being modified. This is the wonder of color manipulation: effects worthy of high-end graphics systems with little effort. Of course, there are limits to what color manipulation can do, but what it does, it does *very* well.

Listing A.1 can perform color paging using either the direct or BIOS approach, depending on the setting of **USE_BIOS**. On my computer, both versions look exactly the same. Experiment for yourself and choose whichever you prefer.

Paging Invisible Pages

One advantage of color paging is that you can change the contents of non-displayed color pages without affecting the display in any way. Manipulating the color pages in this way is much like page flipping-based animation; while you're displaying one color page, you alter another one, then display the other one when it's ready to go. The advantage of this over simply changing the palette is that the user will see only the finished color configuration, not an intermediate state. This approach effectively expands the number of available color pages from 16 to more than 4 million (16 colors per set times 256K choices for each color; that is, every combination of 16 colors possible on the VGA).

Although it's not the main thrust of the code, Listing A.1 does illustrate the technique of changing a non-displayed color page so as to leave the display unaffected while the DAC is being loaded. At the outset of Listing A.1, page 0 is displayed while the DAC locations for pages 1 through 15 are loaded. Once the loading is complete, the program switches away from page 0 to begin cycling through the newly-loaded pages 15 through 1.

That brings us to the end of color paging, but certainly not to the end of color on the VGA. We still haven't looked closely at color cycling, but refer to Chapter 34 for more information.