Chapter 32

Be It Resolved: 360x480

Chapter 32

Taking 256-Color Modes About as Far as the Standard VGA Can Take Them

In the last chapter, we learned how to coax 320×400 256-color resolution out of a standard VGA. At the time, I noted that the VGA was actually capable of supporting 256-color resolutions as high as 360×480, but didn't pursue the topic further, preferring to concentrate on the versatile and easy-to-set 320×400 256-color mode instead.

Some time back I was sent a particularly useful item from John Bridges, a longtime correspondent and an excellent programmer. It was a complete mode set routine for 360×480 256-color mode that he has placed into the public domain. In addition, John wrote, "I also have a couple of freeware (free, but not public domain) utilities out there, including PICEM, which displays PIC, PCX, and GIF images not only in 360×480×256 but also in 640×350×256, 640×400×256, 640×480×256, and 800×600×256 on SuperVGAs."

In this chapter, I'm going to combine John's mode set code with appropriately modified versions of the dot-plot code from Chapter 31 and the line-drawing code that we'll develop in Chapter 35. Together, those routines will make a pretty nifty demo of the capabilities of 360×480 256-color mode.

Extended 256-Color Modes: What's Not to Like?

When last we left 256-color programming, we had found that the standard 256-color mode, mode 13H, which officially offers 320×200 resolution, actually displays 400, not 200, scan lines, with line-doubling used to reduce the effective resolution to 320×200. By tweaking a few of the VGA's mode registers, we converted mode 13H to a true 320×400 256-color mode. As an added bonus, that 320×400 mode supports two graphics pages, a distinct improvement over the single graphics page supported by mode 13H. (We also learned how to get *four* graphics pages at 320×200 resolution, should that be needed.)

I particularly like 320×400 256-color mode for two reasons: It supports two-page graphics, which is very important for animation applications; and it doesn't require changing any of the monitor timing characteristics of the VGA. The mode bits that we changed to produce 320×400 256-color mode are pretty much guaranteed to be the same from one VGA to another, but the monitor-oriented registers are less certain to be constant, especially for VGAs that provide special support for the extended capabilities of various multiscanning monitors.

All in all, those are good arguments for 320×400 256-color mode. However, the counter-argument seems compelling as well—nothing beats higher resolution for producing striking graphics. Given that, and given that John Bridges was kind enough to make his mode set code available, I'm going to look at 360×480 256-color mode next. However, bear in mind that the drawbacks of this mode are the flip side of the strengths of 320×400 256-color mode: Only one graphics page, and direct setting of the monitor-oriented registers. Also, this mode has a peculiar and unique aspect ratio, with 480 pixels (as many as high-resolution mode 12H) vertically and only 360 horizontally. That makes for fairly poor horizontal resolution and sometimes-jagged drawing; on the other hand, the resolution is better in both directions than in mode 13H, and mode 13H itself has an odd aspect ratio, so it seems a bit petty to complain.

The single graphics page isn't a drawback if you don't need page flipping, of course, so there's not much to worry about there: If you need page flipping, don't use this mode. The direct setting of the monitor-oriented registers is another matter altogether.

I don't know how likely this code is to produce problems with clone VGAs in general; however, I did find that I had to put an older Video Seven VRAM VGA into "pure" mode—where it treats the VRAMs as DRAMs and exactly emulates a plain-vanilla IBM VGA—before 360×480 256-color mode would work properly. Now, that particular problem was due to an inherent characteristic of VRAMs, and shouldn't occur on Video Seven's Fastwrite adapter or any other VGA clone. Nonetheless, 360×480 256-color mode is a good deal different from any standard VGA mode, and while the code in this chapter runs perfectly well on all other VGAs in my experience, I can't guarantee its functionality on any particular VGA/monitor combination, unlike 320×400 256-color mode. Mind you, 360×480 256-color mode *should* work on all

VGAs—there are just too many variables involved for me to be certain. Feedback from readers with broad 360×480 256-color experience is welcome.

The above notwithstanding, 360×480 256-color mode offers 64 times as many colors and nearly three times as many pixels as IBM's original CGA color graphics mode, making startlingly realistic effects possible. No mode of the VGA (at least no mode that I know of!), documented or undocumented, offers a better combination of resolution and color; even 320×400 256-color mode has 26 percent fewer pixels.

In other words, 360×480 256-color mode is worth considering—so let's have a look.

360×480 256-Color Mode

I'm going to start by showing you 360×480 256-color mode in action, after which we'll look at how it works. I suspect that once you see what this mode looks like, you'll be more than eager to learn how to use it.

Listing 32.1 contains three C-callable assembly functions. As you would expect, **Set360×480Mode** places the VGA into 360×480 256-color mode. **Draw360×480Dot** draws a pixel of the specified color at the specified location. Finally, **Read360×480Dot** returns the color of the pixel at the specified location. (This last function isn't actually used in the example program in this chapter, but is included for completeness.)

Listing 32.2 contains an adaptation of some C line-drawing code I'll be presenting shortly in Chapter 35. If you're reading this book in serial fashion and haven't gotten there yet, simply take it on faith. If you really *really* need to know how the line-draw code works right *now*, by all means make a short forward call to Chapter 35 and digest it. The line-draw code presented below has been altered to select 360×480 256-color mode, and to cycle through all 256 colors that this mode supports, drawing each line in a different color.

LISTING 32.1 L32-1.ASM

```
; Borland C/C++ tiny/small/medium model-callable assembler
; subroutines to:
;     * Set 360x480 256-color VGA mode
;     * Draw a dot in 360x480 256-color VGA mode
;     * Read the color of a dot in 360x480 256-color VGA mode
;
; Assembled with TASM
;
; The 360x480 256-color mode set code and parameters were provided
; by John Bridges, who has placed them into the public domain.
;
VGA_SEGMENT       equ   0a000h          ;display memory segment
SC_INDEX          equ   3c4h            ;Sequence Controller Index register
GC_INDEX          equ   3ceh            ;Graphics Controller Index register
MAP_MASK          equ   2               ;Map Mask register index in SC
READ_MAP          equ   4               ;Read Map register index in GC
SCREEN_WIDTH            equ   360       ;# of pixels across screen
WORD_OUTS_OK            equ   1         ;set to 0 to assemble for
                                        ; computers that can't handle
                                        ; word outs to indexed VGA registers
```

```
;
_DATA segment    public byte 'DATA'
;
; 360x480 256-color mode CRT Controller register settings.
; (Courtesy of John Bridges.)
;
vptbl   dw      06b00h                  ; horz total
        dw      05901h                  ; horz displayed
        dw      05a02h                  ; start horz blanking
        dw      08e03h                  ; end horz blanking
        dw      05e04h                  ; start h sync
        dw      08a05h                  ; end h sync
        dw      00d06h                  ; vertical total
        dw      03e07h                  ; overflow
        dw      04009h                  ; cell height
        dw      0ea10h                  ; v sync start
        dw      0ac11h                  ; v sync end and protect cr0-cr7
        dw      0df12h                  ; vertical displayed
        dw      02d13h                  ; offset
        dw      00014h                  ; turn off dword mode
        dw      0e715h                  ; v blank start
        dw      00616h                  ; v blank end
        dw      0e317h                  ; turn on byte mode
vpend   label   word
_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
;
_TEXT segment byte public 'CODE'
      assume      cs:_TEXT, ds:_DATA
;
; Sets up 360x480 256-color mode.
; (Courtesy of John Bridges.)
;
; Call as: void Set360By480Mode()
;
; Returns: nothing
;
      public _Set360x480Mode
_Set360x480Mode  proc  near
      push  si                          ;preserve C register vars
      push  di
      mov   ax,12h                      ; start with mode 12h
      int   10h                         ; let the BIOS clear the video memory

      mov   ax,13h                      ; start with standard mode 13h
      int   10h                         ; let the BIOS set the mode
```

```
      mov   dx,3c4h                   ; alter sequencer registers
      mov   ax,0604h                  ; disable chain 4
      out   dx,ax

      mov   ax,0100h                  ; synchronous reset
      out   dx,ax                     ; asserted
      mov   dx,3c2h                   ; misc output
      mov   al,0e7h                   ; use 28 mHz dot clock
      out   dx,al                     ; select it
      mov   dx,3c4h                   ; sequencer again
      mov   ax,0300h                  ; restart sequencer
      out   dx,ax                     ; running again

      mov     dx,3d4h                 ; alter crtc registers

      mov     al,11h                  ; cr11
      out     dx,al                   ; current value
      inc     dx                      ; point to data
      in      al,dx                   ; get cr11 value
      and     al,7fh                  ; remove cr0 -> cr7
      out     dx,al                   ; write protect
      dec     dx                      ; point to index
      cld
      mov     si,offset vptbl
      mov     cx,((offset vpend)-(offset vptbl)) shr 1
@b:   lodsw
      out     dx,ax
      loop    @b
      pop   di                        ;restore C register vars
      pop   si
      ret
_Set360x480Mode  endp
;
; Draws a pixel in the specified color at the specified
; location in 360x480 256-color mode.
;
; Call as: void Draw360x480Dot(int X, int Y, int Color)
;
; Returns: nothing
;
DParms      struc
      dw    ?                         ;pushed BP
      dw    ?                         ;return address
DrawX dw    ?                         ;X coordinate at which to draw
DrawY dw    ?                         ;Y coordinate at which to draw
Color dw    ?                         ;color in which to draw (in the
                                      ; range 0-255; upper byte ignored)
DParms      ends
;
      public _Draw360x480Dot
_Draw360x480Dot  proc  near
      push  bp                        ;preserve caller's BP
      mov   bp,sp                     ;point to stack frame
      push  si                        ;preserve C register vars
      push  di
      mov   ax,VGA_SEGMENT
      mov   es,ax                     ;point to display memory
      mov   ax,SCREEN_WIDTH/4
                                      ;there are 4 pixels at each address, so
                                      ; each 360-pixel row is 90 bytes wide
                                      ; in each plane
```

```
        mul   [bp+DrawY]                ;point to start of desired row
        mov   di,[bp+DrawX]             ;get the X coordinate
        shr   di,1                      ;there are 4 pixels at each address
        shr   di,1                      ; so divide the X coordinate by 4
        add   di,ax                     ;point to the pixel's address
        mov   cl,byte ptr [bp+DrawX]    ;get the X coordinate again
        and   cl,3                      ;get the plane # of the pixel
        mov   ah,1
        shl   ah,cl                     ;set the bit corresponding to the plane
                                        ; the pixel is in
        mov   al,MAP_MASK
        mov   dx,SC_INDEX
        OUT_WORD                        ;set to write to the proper plane for
                                        ; the pixel
        mov   al,byte ptr [bp+Color]    ;get the color
        stosb                           ;draw the pixel
        pop   di                        ;restore C register vars
        pop   si
        pop   bp                        ;restore caller's BP
        ret
_Draw360x480Dot  endp
;
; Reads the color of the pixel at the specified
; location in 360x480 256-color mode.
;
; Call as: int Read360x480Dot(int X, int Y)
;
; Returns: pixel color
;
RParms      struc
        dw    ?                         ;pushed BP
        dw    ?                         ;return address
ReadX dw    ?                           ;X coordinate from which to read
ReadY dw    ?                           ;Y coordinate from which to read
RParms      ends
;
        public _Read360x480Dot
_Read360x480Dot  proc  near
        push  bp                        ;preserve caller's BP
        mov   bp,sp                     ;point to stack frame
        push  si                        ;preserve C register vars
        push  di
        mov   ax,VGA_SEGMENT
        mov   es,ax                     ;point to display memory
        mov   ax,SCREEN_WIDTH/4         ;there are 4 pixels at each address, so
                                        ; each 360-pixel row is 90 bytes wide
                                        ; in each plane

        mul   [bp+DrawY]                ;point to start of desired row
        mov   si,[bp+DrawX]             ;get the X coordinate
        shr   si,1                      ;there are 4 pixels at each address
        shr   si,1                      ; so divide the X coordinate by 4
        add   si,ax                     ;point to the pixel's address
        mov   ah,byte ptr [bp+DrawX]    ;get the X coordinate again
        and   ah,3
                                        ;get the plane # of the pixel
        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
      sub   ah,ah                       ;make the return value a word for C
      pop   di                          ;restore C register vars
      pop   si
      pop   bp                          ;restore caller's BP
      ret
_Read360x480Dot  endp
_TEXT ends
      end
```

LISTING 32.2 L32-2.C

```
 * Sample program to illustrate VGA line drawing in 360x480
 * 256-color mode.
 *
 * Compiled with Borland C/C++.
 *
 * Must be linked with Listing 32.1 with a command line like:
 *
 *    bcc l10-2.c l10-1.asm
 *
 * By Michael Abrash
 */
#include <dos.h>                /* contains geninterrupt */

#define TEXT_MODE       0x03
#define BIOS_VIDEO_INT  0x10
#define X_MAX           360     /* working screen width */
#define Y_MAX           480     /* working screen height */

extern void Draw360x480Dot();
extern void Set360x480Mode();

/*
 * Draws a line in octant 0 or 3 ( |DeltaX| >= DeltaY ).
 * |DeltaX|+1 points are drawn.
 */
void Octant0(X0, Y0, DeltaX, DeltaY, XDirection, Color)
unsigned int X0, Y0;          /* coordinates of start of the line */
unsigned int DeltaX, DeltaY;  /* length of the line */
int XDirection;               /* 1 if line is drawn left to right,
                                 -1 if drawn right to left */
int Color;                    /* color in which to draw line */
{
   int DeltaYx2;
   int DeltaYx2MinusDeltaXx2;
   int ErrorTerm;

   /* Set up initial error term and values used inside drawing loop */
   DeltaYx2 = DeltaY * 2;
   DeltaYx2MinusDeltaXx2 = DeltaYx2 - (int) ( DeltaX * 2 );
   ErrorTerm = DeltaYx2 - (int) DeltaX;

   /* Draw the line */
   Draw360x480Dot(X0, Y0, Color);   /* draw the first pixel */
   while ( DeltaX-- ) {
      /* See if it's time to advance the Y coordinate */
      if ( ErrorTerm >= 0 ) {
         /* Advance the Y coordinate & adjust the error term
            back down */
```

```
            Y0++;
            ErrorTerm += DeltaYx2MinusDeltaXx2;
         } else {
            /* Add to the error term */
            ErrorTerm += DeltaYx2;
         }
         X0 += XDirection;                    /* advance the X coordinate */
         Draw360x480Dot(X0, Y0, Color);       /* draw a pixel */
      }
   }

   /*
    * Draws a line in octant 1 or 2 ( |DeltaX| < DeltaY ).
    * |DeltaY|+1 points are drawn.
    */
   void Octant1(X0, Y0, DeltaX, DeltaY, XDirection, Color)
   unsigned int X0, Y0;          /* coordinates of start of the line */
   unsigned int DeltaX, DeltaY;  /* length of the line */
   int XDirection;               /* 1 if line is drawn left to right,
                                    -1 if drawn right to left */
   int Color;                    /* color in which to draw line */
   {
      int DeltaXx2;
      int DeltaXx2MinusDeltaYx2;
      int ErrorTerm;

      /* Set up initial error term and values used inside drawing loop */
      DeltaXx2 = DeltaX * 2;
      DeltaXx2MinusDeltaYx2 = DeltaXx2 - (int) ( DeltaY * 2 );
      ErrorTerm = DeltaXx2 - (int) DeltaY;

      Draw360x480Dot(X0, Y0, Color);       /* draw the first pixel */
      while ( DeltaY-- ) {
         /* See if it's time to advance the X coordinate */
         if ( ErrorTerm >= 0 ) {
            /* Advance the X coordinate & adjust the error term
               back down */
            X0 += XDirection;
            ErrorTerm += DeltaXx2MinusDeltaYx2;
         } else {
            /* Add to the error term */
            ErrorTerm += DeltaXx2;
         }
         Y0++;       /* advance the Y coordinate */
         Draw360x480Dot(X0, Y0,Color);     /* draw a pixel */
      }
   }

   /*
    * Draws a line on the EGA or VGA.
    */
   void EVGALine(X0, Y0, X1, Y1, Color)
   int X0, Y0;       /* coordinates of one end of the line */
   int X1, Y1;       /* coordinates of the other end of the line */
   unsigned char Color;   /* color in which to draw line */
   {
      int DeltaX, DeltaY;
      int Temp;
```

```
   /* Save half the line-drawing cases by swapping Y0 with Y1
      and X0 with X1 if Y0 is greater than Y1. As a result, DeltaY
      is always > 0, and only the octant 0-3 cases need to be
      handled. */
   if ( Y0 > Y1 ) {
      Temp = Y0;
      Y0 = Y1;
      Y1 = Temp;
      Temp = X0;
      X0 = X1;
      X1 = Temp;
   }

   /* Handle as four separate cases, for the four octants in which
      Y1 is greater than Y0 */
   DeltaX = X1 - X0;     /* calculate the length of the line
                            in each coordinate */
   DeltaY = Y1 - Y0;
   if ( DeltaX > 0 ) {
      if ( DeltaX > DeltaY ) {
         Octant0(X0, Y0, DeltaX, DeltaY, 1, Color);
      } else {
         Octant1(X0, Y0, DeltaX, DeltaY, 1, Color);
      }
   } else {
      DeltaX = -DeltaX;              /* absolute value of DeltaX */
      if ( DeltaX > DeltaY ) {
         Octant0(X0, Y0, DeltaX, DeltaY, -1, Color);
      } else {
         Octant1(X0, Y0, DeltaX, DeltaY, -1, Color);
      }
   }
}

/*
 * Subroutine to draw a rectangle full of vectors, of the
 * specified length and in varying colors, around the
 * specified rectangle center.
 */
void VectorsUp(XCenter, YCenter, XLength, YLength)
int XCenter, YCenter;   /* center of rectangle to fill */
int XLength, YLength;   /* distance from center to edge
                           of rectangle */
{
   int WorkingX, WorkingY, Color = 1;

   /* Lines from center to top of rectangle */
   WorkingX = XCenter - XLength;
   WorkingY = YCenter - YLength;
   for ( ; WorkingX < ( XCenter + XLength ); WorkingX++ )
      EVGALine(XCenter, YCenter, WorkingX, WorkingY, Color++);

   /* Lines from center to right of rectangle */
   WorkingX = XCenter + XLength - 1;
   WorkingY = YCenter - YLength;
   for ( ; WorkingY < ( YCenter + YLength ); WorkingY++ )
      EVGALine(XCenter, YCenter, WorkingX, WorkingY, Color++);

   /* Lines from center to bottom of rectangle */
   WorkingX = XCenter + XLength - 1;
   WorkingY = YCenter + YLength - 1;
```

```
   for ( ; WorkingX >= ( XCenter - XLength ); WorkingX-- )
      EVGALine(XCenter, YCenter, WorkingX, WorkingY, Color++);

   /* Lines from center to left of rectangle */
   WorkingX = XCenter - XLength;
   WorkingY = YCenter + YLength - 1;
   for ( ; WorkingY >= ( YCenter - YLength ); WorkingY-- )
      EVGALine(XCenter, YCenter, WorkingX, WorkingY, Color++);
}

/*
 * Sample program to draw four rectangles full of lines.
 */
void main()
{
   char temp;

   Set360x480Mode();

   /* Draw each of four rectangles full of vectors */
   VectorsUp(X_MAX / 4, Y_MAX / 4, X_MAX / 4, Y_MAX / 4, 1);
   VectorsUp(X_MAX * 3 / 4, Y_MAX / 4, X_MAX / 4, Y_MAX / 4, 2);
   VectorsUp(X_MAX / 4, Y_MAX * 3 / 4, X_MAX / 4, Y_MAX / 4, 3);
   VectorsUp(X_MAX * 3 / 4, Y_MAX * 3 / 4, X_MAX / 4, Y_MAX / 4, 4);

   /* Wait for the enter key to be pressed */
   scanf("%c", &temp);

   /* Back to text mode */
   _AX = TEXT_MODE;
   geninterrupt(BIOS_VIDEO_INT);
}
```

The first thing you'll notice when you run this code is that the speed of 360×480 256-color mode is pretty good, especially considering that most of the program is implemented in C.

Drawing in 360×480 256-color mode can sometimes actually be faster than in the 16-color modes, because the byte-per-pixel display memory organization of 256-color mode eliminates the need to read display memory before writing to it in order to isolate individual pixels coexisting within a single byte. In addition, 360×480 256-color mode is a variant of Mode X, which we'll encounter in detail in Chapter 47, and supports all the high-performance features of Mode X.

The second thing you'll notice is that exquisite shading effects are possible in 360×480 256-color mode; adjacent lines blend together remarkably smoothly, even with the default palette. The VGA allows you to select your 256 colors from a palette of 256K, so you could, if you wished, set up the colors to produce still finer shading albeit with fewer distinctly different colors available. For more on this and related topics, see the coverage of palette reprogramming that begins in the next chapter.

The one thing you may not notice right away is just how much detail is visible on the screen, because the blending of colors tends to obscure the superior resolution of

this mode. Each of the four rectangles displayed measures 180 pixels horizontally by 240 vertically. Put another way, each *one* of those rectangles has two-thirds as many pixels as the entire mode 13H screen; in all, 360×480 256-color mode has 2.7 times as many pixels as mode 13H! As mentioned above, the resolution is unevenly distributed, with vertical resolution matching that of mode 12H but horizontal resolution barely exceeding that of mode 13H—but resolution is hot stuff, no matter how it's laid out, and 360×480 256-color mode has the highest 256-color resolution you're ever likely to see on a standard VGA. (SuperVGAs are quite another matter—but when you *require* a SuperVGA you're automatically excluding what might be a significant chunk of the market for your code.)

Now that we've seen the wonders of which our new mode is capable, let's take the time to understand how it works.

How 360×480 256-Color Mode Works

In describing 360×480 256-color mode, I'm going to assume that you're familiar with the discussion of 320×400 256-color mode in the last chapter. If not, go back to that chapter and read it; the two modes have a great deal in common, and I'm not going to bore you by repeating myself when the goods are just a few page flips (the paper kind) away.

360×480 256-color mode is essentially 320×400 256-color mode, but stretched in both dimensions. Let's look at the vertical stretching first, since that's the simpler of the two.

480 Scan Lines per Screen: A Little Slower, But No Big Deal

There's nothing unusual about 480 scan lines; standard modes 11H and 12H support that vertical resolution. The number of scan lines has nothing to do with either the number of colors or the horizontal resolution, so converting 320×400 256-color mode to 320×480 256-color mode is a simple matter of reprogramming the VGA's vertical control registers—which control the scan lines displayed, the vertical sync pulse, vertical blanking, and the total number of scan lines—to the 480-scan-line settings, and setting the polarities of the horizontal and vertical sync pulses to tell the monitor to adjust to a 480-line screen.

Switching to 480 scan lines has the effect of slowing the screen refresh rate. The VGA always displays at 70 Hz *except* in 480-scan-line modes; there, due to the time required to scan the extra lines, the refresh rate slows to 60 Hz. (VGA monitors always scan at the same rate horizontally; that is, the distance across the screen covered by the electron beam in a given period of time is the same in all modes. Consequently, adding extra lines per frame requires extra time.) 60 Hz isn't *bad*—that's the only refresh rate the EGA ever supported, and the EGA was the industry standard in its time—but it does tend to flicker a little more and so is a little harder on the eyes than 70 Hz.

360 Pixels per Scan Line: No Mean Feat

Converting from 320 to 360 pixels per scan line is more difficult than converting from 400 to 480 scan lines per screen. None of the VGA's graphics modes supports 360 pixels across the screen, or anything like it; the standard choices are 320 and 640 pixels across. However, the VGA *does* support the horizontal resolution we seek—360 pixels—in 40-column *text* mode.

Unfortunately, the register settings that select those horizontal resolutions aren't directly transferable to graphics mode. Text modes display 9 dots (the width of one character) for each time information is fetched from display memory, while graphics modes display just 4 or 8 dots per display memory fetch. (Although it's a bit confusing, it's standard terminology to refer to the interval required for one display memory fetch as a "character," and I'll follow that terminology from now on.) Consequently, both modes display either 40 or 80 characters per scan line; the only difference is that text modes display more pixels per character. Given that graphics modes *can't* display 9 dots per character (there's only enough information for eight 16-color pixels or four 256-color pixels in each memory fetch, and that's that), we'd seem to be at an impasse.

The key to solving this problem lies in recalling that the VGA is designed to drive a monitor that sweeps the electron beam across the screen at exactly the same speed, no matter what mode the VGA is in. If the monitor always sweeps at the same speed, how does the VGA manage to display both 640 pixels across the screen (in high-resolution graphics modes) and 720 pixels across the screen (in 80-column text modes)? Good question indeed—and the answer is that the VGA has not one but *two* clocks on board, and one of those clocks is just sufficiently faster than the other clock so that an extra 80 (or 40) pixels can be displayed on each scan line.

In other words, there's a slow clock (about 25 MHz) that's usually used in graphics modes to get 640 (or 320) pixels on the screen during each scan line, and a second, fast clock (about 28 MHz) that's usually used in text modes to crank out 720 (or 360) pixels per scan line. In particular, 320×400 256-color mode uses the 25 MHz clock.

I'll bet that you can see where I'm headed: We can switch from the 25 MHz clock to the 28 MHz clock in 320×480 256-color mode in order to get more pixels. It takes two clocks to produce one 256-color pixel, so we'll get 40 rather than 80 extra pixels by doing this, bringing our horizontal resolution to the desired 360 pixels.

Switching horizontal resolutions sounds easy, doesn't it? Alas, it's not. There's no standard VGA mode that uses the 28 MHz clock to draw 8 rather than 9 dots per character, so the timing parameters have to be calculated from scratch. John Bridges has already done that for us, but I want you to appreciate that producing this mode took some work. The registers controlling the total number of characters per scan line, the number of characters displayed, the horizontal sync pulse, horizontal blanking, the offset from the start of one line to the start of the next, and the clock speed all have to be

altered in order to set up 360×480 256-color mode. The function **Set360×480Mode** in Listing 32.1 does all that, and sets up the registers that control vertical resolution, as well.

Once all that's done, the VGA is in 360×480 mode, awaiting our every high-resolution 256-color graphics whim.

Accessing Display Memory in 360×480 256-Color Mode

Setting up for 360×480 256-color mode proved to be quite a task. Is drawing in this mode going to be as difficult?

No. In fact, if you know how to draw in 320×400 256-color mode, you already know how to draw in 360×480 256-color mode; the conversion between the two is a simple matter of changing the working screen width from 320 pixels to 360 pixels. In fact, if you were to take the 320×400 256-color pixel reading and pixel writing code from Chapter 31 and change the **SCREEN_WIDTH** equate from 320 to 360, those routines would work perfectly in 360×480 256-color mode.

The organization of display memory in 360×480 256-color mode is almost exactly the same as in 320×400 256-color mode, which we covered in detail in the last chapter. However, as a quick refresher, each byte of display memory controls one 256-color pixel, just as in mode 13H. The VGA is reprogrammed by the mode set so that adjacent pixels lie in adjacent planes of display memory. Look back to Figure 31.1 in the last chapter to see the organization of the first few pixels on the screen; the bytes controlling those pixels run cross-plane, advancing to the next address only every fourth pixel. The address of the pixel at screen coordinate (x,y) is

$address = ((y*360)+x)/4$

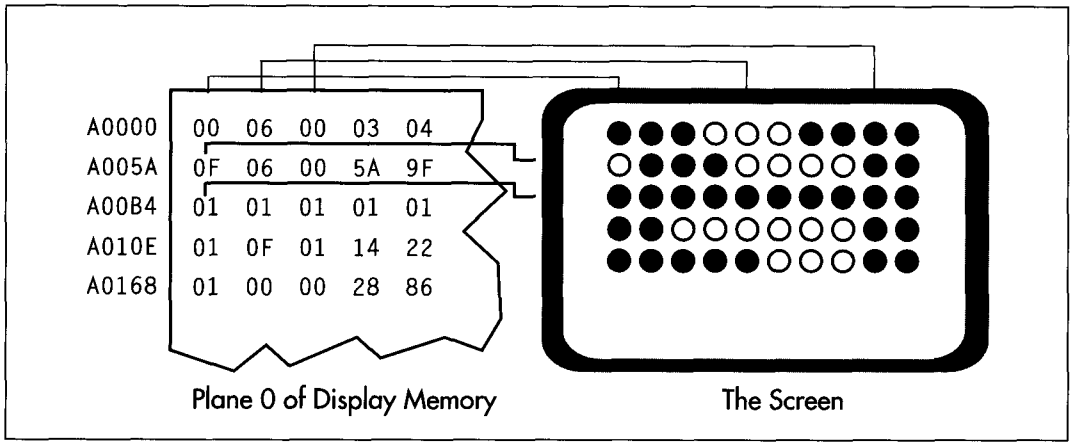
and the plane of a given pixel is:

$plane = x$ modulo 4

A new scan line starts every 360 pixels, or 90 bytes, as shown in Figure 32.1. This is the major programming difference between the 360×480 and 320×400 256-color modes; in the 320×400 mode, a new scan line starts every 80 bytes.

The other programming difference between the two modes is that the area of display memory mapped to the screen is longer in 360×480 256-color mode, which is only common sense given that there are more pixels in that mode. The exact amount of memory required in 360×480 256-color mode is 360 times 480 = 172,800 bytes. That's more than half of the VGA's 256 Kb memory complement, so page-flipping is out; however, there's no reason you couldn't use that extra memory to create a virtual screen larger than 360×480, around which you could then scroll, if you wish.

That's really all there is to drawing in 360×480 256-color mode. From a programming perspective, this mode is no more complicated than 320×400 256-color mode once the mode set is completed, and should be capable of good performance given some clever coding. It's not particular straightforward to implement bitblt, block



Pixel organization in 360×480 256-color mode.
Figure 32.1

move, or fast line-drawing code for any of the extended 256-color modes, but it can be done—and it's worth the trouble. Even the small taste we've gotten of the capabilities of these modes shows that they put the traditional CGA, EGA, and generally even VGA modes to shame.

There's more and better to come, though; in later chapters, we'll return to high-resolution 256-color programming in a big way, by exploring the tremendous potential of these modes for real time 2-D and 3-D animation.