Mode X Marks the Latch
In the previous chapter, I introduced you to what I call Mode X, an undocumented 320×240 256-color mode of the VGA. Mode X is distinguished from mode 13H, the documented 320×200 256-color VGA mode, in that it supports page flipping, makes off-screen memory available, has square pixels, and, above all, lets you use the VGA's hardware to increase performance by as much as four times. (Of course, those four times come at the cost of more complex and demanding programming, to be sure—but end users care about results, not how hard the code was to write, and Mode X delivers results in a big way.) In the previous chapter we saw how the VGA's plane-oriented hardware can be used to speed solid fills. That's a nice technique, but now we're going to move up to the big guns—the VGA latches.

The VGA has four latches, one for each plane of display memory. Each latch stores exactly one byte, and that byte is always the last byte read from the corresponding plane of display memory, as shown in Figure 48.1. Furthermore, whenever a given address in display memory is read, all four planes' bytes at that address are read and stored in the corresponding latches, regardless of which plane supplied the byte returned to the CPU (as determined by the Read Map register). As with so much else about the VGA, the above will make little sense to VGA neophytes, but the important point is this: By reading one display memory byte, 4 bytes—one from each plane—can be loaded into the latches at once. Any or all of those 4 bytes can then be written anywhere in display memory with a single byte-sized write, as shown in Figure 48.2.
The value 49, from plane 1, is read by the CPU

All four latches are loaded from the corresponding planes by every display memory read

How the VGA latches are loaded.

Figure 48.1

The value OFFh is written by the CPU

Writing 4 bytes to display memory in a single operation.

Figure 48.2
The upshot is that the latches make it possible to copy data around from one part of display memory to another, 32 bits (four pixels) at a time—four times as fast as normal. (Recall from the previous chapter that in Mode X, pixels are stored one per byte, with four pixels in a row stored in successive planes at the same address, one pixel per plane.) However, any one latch can only be loaded from and written to the corresponding plane, so an individual latch can only work with every fourth pixel on the screen; the latch for plane 0 can work with pixels 0, 4, 8..., the latch for plane 1 with pixels 1, 5, 9..., and so on.

The latches aren’t intended for use in 256-color mode—they were designed to allow individual bits of display memory to be modified in 16-color mode—but they are nonetheless very useful in Mode X, particularly for patterned fills and screen-to-screen copies, including scrolls. Patterned filling is a good place to start, because patterns are widely used in windowing environments for desktops, window backgrounds, and scroll bars, and for textures and color dithering in drawing and game software.

Fast Mode X fills using patterns that are four pixels in width can be performed by drawing the pattern once to the four pixels at any one address in display memory, reading that address to load the pattern into the latches, setting the Bit Mask register to 0 to specify that all bits drawn to display memory should come from the latches, and then performing the fill pretty much as we did in the previous chapter—except that each line of the pattern must be loaded into the latches before the corresponding scan line on the screen is filled. Listings 48.1 and 48.2 together demonstrate a variety of fast Mode X four-by-four pattern fills. (The mode set function called by Listing 48.1 is from the previous chapter’s listings.)

**LISTING 48.1 L48-1.C**

```c
/* Program to demonstrate Mode X (320x240, 256 colors) patterned rectangle fills by filling the screen with adjacent 80x60 rectangles in a variety of patterns. Tested with Borland C++ in C compilation mode and the small model */
#include <conio.h>
#include <dos.h>

void Set320x240Mode(void);
void FillPatternX(int, int, int, int, unsigned int, char*);

/* 16 4x4 patterns */
static char Patt0[]=\{10,0,10,0,0,10,0,10,0,10,0,10,0,10,0,10\};
static char Patt1[]=\{9,0,0,0,9,0,0,0,9,0,0,0,0,0,0,9\};
static char Patt2[]=\{5,0,0,0,0,5,0,5,0,0,0,0,0,0,0,5\};
static char Patt3[]=\{14,0,0,14,0,14,0,14,0,14,0,14,0,14,0,14\};
static char Patt4[]=\{15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15\};
static char Patt5[]=\{12,12,12,12,6,6,6,12,6,6,6,12,6,6,6,12\};
static char Patt6[]=\{80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,80\};
static char Patt7[]=\{78,78,78,78,80,80,80,80,82,82,82,82,82,82,82,82\};
static char Patt8[]=\{78,80,82,84,80,82,82,82,82,82,82,82,82,82,82,82\};
static char Patt9[]=\{78,80,82,84,78,80,82,82,82,82,82,82,82,82,82,82\};
static char Patt10[]=\{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15\};
static char Patt11[]=\{0,1,2,3,0,1,2,3,0,1,2,3,0,1,2,3\};
static char Patt12[]=\{14,14,9,9,14,9,9,9,14,9,9,9,14,9,9,14\};
static char Patt13[]=\{15,8,8,8,15,15,15,15,15,8,15,15,15,8,15,8,8\};
```

Mode X Marks the Latch 899
static char Patt14[] = {3,3,3,3,3,7,3,3,7,3,3,3,3};
static char Patt15[] = {0,0,0,0,64,0,0,0,0,0,0,0,89};

/* Table of pointers to the 16 4x4 patterns with which to draw */
static char* PattTable[] = {Patt0, Patt1, Patt2, Patt3, Patt4, Patt5, Patt6, 
Patt7, Patt8, Patt9, Patt10, Patt11, Patt12, Patt13, Patt14, Patt15};

void main() {
    int i, j;
    union REGS regset;

    Set320x240Mode();
    for (j = 0; j < 4; j++) {
        for (i = 0; i < 4; i++) {
            FillPatternX(i*80, j*60, i*80+80, j*60+60, 0, PattTable[j*4+1]);
        }
    }
    getch();
    regset.x.ax = 0x0003; /* switch back to text mode and done */
    int86(0x10, &regset, &regset);
}

LISTING 48.2 L48-2.ASM

; Mode X (320x240, 256 colors) rectangle 4x4 pattern fill routine.
; Upper left corner of pattern is always aligned to a multiple-of-4 row and column. Works on all VGAs. Uses approach of copying the pattern to off-screen display memory, then loading the latches with the pattern for each scan line and filling each scan line four pixels at a time. Fills up to but not including the column at EndX and the row at EndY. No clipping is performed. All ASM code tested with TASM. C near-callable as:

; void FillPatternX(int StartX, int StartY, int EndX, int EndY, unsigned int PageBase, char* Pattern);

SC_INDEX equ 03c4h
MAP_MASK equ 02h
GC_INDEX equ 03ceh
BIT_MASK equ 08h
PATTERN_BUFFER equ 0ffch
SCREEN_SEG equ 0a000h
SCREEN_WIDTH equ 80

parms struc
dw 2 dup (?)
StartX dw ?
StartY dw ?
EndX dw ?
EndY dw ?
PageBase dw ?
Pattern dw ?
parms ends

NextScanOffset equ -2
RectAddrWidth equ -4
Height equ -6
STACK_FRAME_SIZE equ 6
.model small
.data
: Plane masks for clipping left and right edges of rectangle.
LeftClipPlaneMask db 00fh,00eh,00ch,008h
RightClipPlaneMask db 00fh,001h,003h,007h
.code
public _FillPatternX
_FillPatternX proc near
push bp
; preserve caller's stack frame
mov bp,sp
; point to local stack frame
sub sp,STACK_FRAME_SIZE
; allocate space for local vars
push si
; preserve caller's register variables
push di

clcd
mov ax,SCREEN_SEG
; point ES to display memory
mov es,ax

mov si,[bp+Pattern]
; copy pattern to display memory buffer
mov di,PATTERN_BUFFER
; point ES:DI to pattern buffer
mov dx,SC_INDEX
; point Sequencer Controller Index to
mov al,MAP_MASK
; Map Mask
out dx,al
inc dx
; point to SC Data register
mov cx,4
; 4 pixel quadruplets in pattern
DownloadPatternLoop:
mov al,1
; select plane 0 for writes
movsb
di
; copy over next plane 0 pattern pixel
dec di
; stay at same address for next plane
mov al,2
; select plane 1 for writes
out dx,al
movsb
di
; copy over next plane 1 pattern pixel
dec di
; stay at same address for next plane
mov al,4
; select plane 2 for writes
out dx,al
movsb
di
; copy over next plane 2 pattern pixel
dec di
; stay at same address for next plane
mov al,8
; select plane 3 for writes
out dx,al
movsb
loop DownloadPatternLoop

mov dx,GC_INDEX
; set the bit mask to select all bits
mov ax,00000h+BIT_MASK
; from the latches and none from
out dx,ax
; the CPU, so that we can write the
; latch contents directly to memory
mov ax,[bp+StartY]
; top rectangle scan line
mov si,ax
and si,011b
; top rect scan line modulo 4
add si,PATTERN_BUFFER
; maps to top line of rect to draw
mov dx,SCREEN_WIDTH
mul dx
mov dx,[bp+StartX]
mov bx,dx
shrd d1.1
shr d1.1
add d1,ax
; X/4 = offset of first rectangle pixel in scan

_mode X Marks the Latch 901
add di,[bp+PageBase] ; offset of first rectangle pixel in display memory
and bx,0003h ; look up left edge plane mask
mov ah,LeftClipPlaneMask[bx] ; to clip
mov bx,[bp+EndX]
and bx,0003h ; look up right edge plane
mov al,RightClipPlaneMask[bx] ; mask to clip
mov bx,ax ; put the masks in BX
mov cx,[bp+EndX] ; calculate # of addresses across rect
mov cx,ax
jle FillDone ; skip if 0 or negative width
dec cx
and ax,not 011b
sub cx,ax
shr cx,1
shr cx,1 ; # of addresses across rectangle to fill - 1
jnez MasksSet ; there's more than one pixel to draw
and bh,bl ; there's only one pixel, so combine the left- and right-edge clip masks
MasksSet:
    mov ax,[bp+EndY]
sub ax,[bp+StartY] ; AX = height of rectangle
jle FillDone ; skip if 0 or negative height
mov [bp+Height].ax ; distance from end of one scan line to start
mov ax,SCREEN_WIDTH
sub ax,ax ; of next
mov [bp+NextScanOffset].ax ; remember width in addresses - 1
mov [bp+RectAddrWidth].cx ; point to Sequence Controller Data reg
mov dx.SC_INDEX+1 ; (SC Index still points to Map Mask)
FillRowsLoop:
    mov cx,[bp+RectAddrWidth] ; width across - 1
    mov al.es:[si] ; read display memory to latch this scan line's pattern
    inc si ; point to next pattern scan line, wrapping
    jnz short NoWrap ; back to start of the pattern if
    sub si,4 ; we've run off the end
NoWrap:
    mov al,bh ; put left-edge clip mask in AL
    out dx.al ; set the left-edge plane (clip) mask
    stosb ; draw the left edge (pixels come from latches; value written by CPU doesn't matter)
    dec cx ; count off left edge address
    js FillLoopBottom ; that's the only address
    jz DoRightEdge ; there are only two addresses
    mov al,00fh ; middle addresses are drawn 4 pixels at a pop
    out dx.al ; set the middle pixel mask to no clip
    rep stosb ; draw the middle addresses four pixels apiece (from latches; value written doesn't matter)
DoRightEdge:
    mov al,b1 ; put right-edge clip mask in AL
    out dx.al ; set the right-edge plane (clip) mask
    stosb ; draw the right edge (from latches; value written doesn't matter)
FillLoopBottom:
    add di,[bp+NextScanOffset] ; point to the start of the next scan line of the rectangle
Four-pixel-wide patterns are more useful than you might imagine. There are actually 2128 possible patterns (16 pixels, each with 28 possible colors); that set is certainly large enough for most color-dithering purposes, and includes many often-used patterns, such as halftones, diagonal stripes, and crosshatches.

Furthermore, eight-wide patterns, which are widely used, can be drawn with two passes, one for each half of the pattern. This principle can in fact be extended to patterns of arbitrary multiple-of-four widths. (Widths that aren't multiples of four are considerably more difficult to handle, because the latches are four pixels wide; one possible solution is expanding such patterns via repetition until they are multiple-of-four widths.)

**Allocating Memory in Mode X**

Listing 48.2 raises some interesting questions about the allocation of display memory in Mode X. In Listing 48.2, whenever a pattern is to be drawn, that pattern is first drawn in its entirety at the very end of display memory; the latches are then loaded from that copy of the pattern before each scan line of the actual fill is drawn. Why this double copying process, and why is the pattern stored in that particular area of display memory?

The double copying process is used because it's the easiest way to load the latches. Remember, there's no way to get information directly from the CPU to the latches; the information must first be written to some location in display memory, because the latches can be loaded *only* from display memory. By writing the pattern to off-screen memory, we don't have to worry about interfering with whatever is currently displayed on the screen.

As for why the pattern is stored exactly where it is, that's part of a master memory allocation plan that will come to fruition in the next chapter, when I implement a Mode X animation program. Figure 48.3 shows this master plan; the first two pages of memory (each 76,800 pixels long, spanning 19,200 addresses—that is, 19,200 pixel quadruplets—in display memory) are reserved for page flipping, the next page of memory (also 76,800 pixels long) is reserved for storing the background (which is
A useful Mode X display memory layout.

Figure 48.3

used to restore the holes left after images move), the last 16 pixels (four addresses) of display memory are reserved for the pattern buffer, and the remaining 31,728 pixels (7,932 addresses) of display memory are free for storage of icons, images, temporary buffers, or whatever.

This is an efficient organization for animation, but there are certainly many other possible setups. For example, you might choose to have a solid-colored background, in which case you could dispense with the background page (instead using the solid rectangle fill routine to replace the background after images move), freeing up another 76,800 pixels of off-screen storage for images and buffers. You could even eliminate page-flipping altogether if you needed to free up a great deal of display memory. For example, with enough free display memory it is possible in Mode X to create a virtual bitmap three times larger than the screen, with the screen becoming a scrolling window onto that larger bitmap. This technique has been used to good effect in a number of animated games, with and without the use of Mode X.
Copying Pixel Blocks within Display Memory

Another fine use for the latches is copying pixels from one place in display memory to another. Whenever both the source and the destination share the same nibble alignment (that is, their start addresses modulo four are the same), it is not only possible but quite easy to use the latches to copy four pixels at a time. Listing 48.3 shows a routine that copies via the latches. (When the source and destination do not share the same nibble alignment, the latches cannot be used because the source and destination planes for any given pixel differ. In that case, you can set the Read Map register to select a source plane and the Map Mask register to select the corresponding destination plane. Then, copy all pixels in that plane, repeating for all four planes.)

Although copying through the latches is, in general, a speedy technique, especially on slower VGAs, it's not always a win. Reading video memory tends to be quite a bit slower than writing, and on a fast VLB or PCI adapter, it can be faster to copy from main memory to display memory than it is to copy from display memory to display memory via the latches.

**LISTING 48.3 L48-3.ASM**

: Mode X (320x240, 256 colors) display memory to display memory copy
: routine. Left edge of source rectangle modulo 4 must equal left edge
: of destination rectangle modulo 4. Works on all VGAs. Uses approach
: of reading 4 pixels at a time from the source into the latches, then
: writing the latches to the destination. Copies up to but not
: including the column at SourceEndX and the row at SourceEndY. No
: clipping is performed. Results are not guaranteed if the source and
: destination overlap. C near-callable as:
:
: void CopyScreenToScreenX(int SourceStartX, int SourceStartY,
: int SourceEndX, int SourceEndY, int DestStartX,
: int DestStartY, unsigned int SourcePageBase,
: unsigned int DestPageBase, int SourceBitmapWidth,
: int DestBitmapWidth);

SC_INDEX equ 03c4h ;Sequence Controller Index register port
MAP_MASK equ 02h ;index in SC of Map Mask register
GC_INDEX equ 03ceh ;Graphics Controller Index register port
BIT_MASK equ 08h ;index in GC of Bit Mask register
SCREEN_SEG equ Oa000h ;segment of display memory in Mode X

parms struc
dw 2 dup (?) ;pushed BP and return address
SourceStartX dw ? ;X coordinate of upper-left corner of source
SourceStartY dw ? ;Y coordinate of upper-left corner of source
SourceEndX dw ? ;X coordinate of lower-right corner of source
; (the row at SourceEndX is not copied)
SourceEndY dw ? ;Y coordinate of lower-right corner of source
; (the column at SourceEndY is not copied)
DestStartX dw ? ;X coordinate of upper-left corner of dest
DestStartY dw ? ;Y coordinate of upper-left corner of dest
SourcePageBase dw ? ;base offset in display memory of page in
; which source resides
DestPageBase dw ? ;base offset in display memory of page in
; which dest resides

Mode X Marks the Latch 905
SourceBitmapWidth  dw  ? ; # of pixels across source bitmap
DestBitmapWidth  dw  ? ; # of pixels across dest bitmap

parms  ends

SourceNextScanOffset  equ  -2 ; local storage for distance from end of
DestNextScanOffset  equ  -4 ; one source scan line to start of next
RectAddrWidth  equ  -6 ; local storage for distance from end of
Height  equ  -8 ; one dest scan line to start of next
STACK_FRAME_SIZE  equ  8 ; local storage for address width of rectangle
 ; local storage for height of rectangle

.model  small
.data
; Plane masks for clipping left and right edges of rectangle.
LeftClipPlaneMask  db  00fh,00eh,00ch,008h
RightClipPlaneMask  db  00fh,001h,003h,007h
.code
public  _CopyScreenToScreenX
_CopyScreenToScreenX  proc  near
    push  bp               ; preserve caller's stack frame
    mov  bp,sp             ; point to local stack frame
    sub  sp,STACK_FRAME_SIZE ; allocate space for local vars
    push  si               ; preserve caller's register variables
    push  di
    push  ds
    cld

    mov  dx,GC_INDEX       ; set the bit mask to select all bits
    mov  ax,000000h+BIT_MASK ; from the latches and none from
    out  dx,ax             ; the CPU, so that we can write the
    mov  ax,SCREEN_SEG    ; latch contents directly to memory
    mov  es,ax             ; point ES to display memory
    mov  ax,[bp+DestBitmapWidth] ; convert to width in addresses
    shr  ax,1
    shr  ax,1
    mull  [bp+DestStartY]  ; top dest rect scan line
    mov  di,[bp+DestStartX] ; X/4 - offset of first dest rect pixel in
    shr  di,1
    shr  di,1
    add  di,ax            ; scan line
    add  di,[bp+DestPageBase] ; offset of first dest rect pixel in page
    add  di,[bp+DestBitmapWidth] ; in display memory
    mov  ax,[bp+SourceBitmapWidth]
    shr  ax,1
    shr  ax,1
    mull  [bp+SourceStartY] ; top source rect scan line
    mov  si,[bp+SourceStartX]
    mov  bx,si
    shr  si,1
    shr  si,1
    add  si,ax            ; X/4 - offset of first source rect pixel in
    add  si,[bp+SourcePageBase] ; scan line
    add  si,[bp+SourceBitmapWidth] ; offset of first source rect
    pixel in display memory
and  bx,0003h ; look up left edge plane mask
    mov  ah,LeftClipPlaneMask[bx] ; to clip
    mov  bx,[bp+SourceEndX]

 petitions for the next page.
and bx, 0003h : look up right-edge plane
mov al, RightClipPlaneMask[bx] : mask to clip
mov bx, ax : put the masks in BX
mov cx, [bp + SourceEndX] : calculate # of addresses across
mov ax, [bp + SourceStartX] : rect
cmp cx, ax
jle CopyDone : skip if 0 or negative width
dec cx
and ax, not 01h
sub cx, ax
shr cx, 1
shr cx, 1
:j # of addresses across rectangle to copy - 1
jnz MasksSet : there's more than one address to draw
and bh, bl : there's only one address, so combine the
: left- and right-edge clip masks

MasksSet:
mov ax, [bp + SourceEndY]
sub cx, [bp + SourceStartY] : AX = height of rectangle
jle CopyDone : skip if 0 or negative height
mov [bp + Height].ax
mov ax, [bp + DestBitmapWidth]
shr ax, 1
shr ax, 1
sub ax, cx
dec ax
mov [bp + DestNextScanOffset].ax
mov ax, [bp + SourceBitmapWidth]
shr ax, 1
shr ax, 1
sub ax, cx
dec ax
mov [bp + SourceNextScanOffset].ax
mov [bp + RectAddrWidth].cx
: remember width in addresses - 1

;-----------------------------BUG FIX
mov dx, SC_INDEX
mov al, MAP_MASK
out dx, al : point SC Index reg to Map Mask
inc dx : point to SC Data reg
;-----------------------------BUG FIX
mov ax, es
mov dx, ax : DS = ES = screen segment for MOVs

CopyRowsLoop:
mov cx, [bp + RectAddrWidth] : width across - 1
mov al, bh : put left-edge clip mask in AL
out dx, al : set the left-edge plane (clip) mask
movsb : copy the left edge (pixels go through
: latches)
dec cx : count off left edge address
js CopyLoopBottom : that's the only address
jz DoRightEdge : there are only two addresses
mov al, 00fh : middle addresses are drawn 4 pixels at a pop
out dx, al : set the middle pixel mask to no clip
rep movsb : draw the middle addresses four pixels apiece
: (pixels copied through latches)

DoRightEdge:
mov al, bl : put right-edge clip mask in AL
out dx, al : set the right-edge plane (clip) mask
movsb : draw the right edge (pixels copied through
: latches)
Listing 48.3 has an important limitation: It does not guarantee proper handling when the source and destination overlap, as in the case of a downward scroll, for example. Listing 48.3 performs top-to-bottom, left-to-right copying. Downward scrolls require bottom-to-top copying; likewise, rightward horizontal scrolls require right-to-left copying. As it happens, my intended use for Listing 48.3 is to copy images between off-screen memory and on-screen memory, and to save areas under pop-up menus and the like, so I don’t really need overlap handling—and I do really need to keep the complexity of this discussion down. However, you will surely want to add overlap handling if you plan to perform arbitrary scrolling and copying in display memory.

Now that we have a fast way to copy images around in display memory, we can draw icons and other images as much as four times faster than in mode 13H, depending on the speed of the VGA’s display memory. (In case you’re worried about the nibble-alignment limitation on fast copies, don’t be; I’ll address that fully in due time, but the secret is to store all four possible rotations in off-screen memory, then select the correct one for each copy.) However, before our fast display memory-to-display memory copy routine can do us any good, we must have a way to get pixel patterns from system memory into display memory, so that they can then be copied with the fast copy routine.

Copying to Display Memory

The final piece of the puzzle is the system memory to display-memory-copy-routine shown in Listing 48.4. This routine assumes that pixels are stored in system memory in exactly the order in which they will ultimately appear on the screen; that is, in the same linear order that mode 13H uses. It would be more efficient to store all the pixels for one plane first, then all the pixels for the next plane, and so on for all four planes, because many OUTs could be avoided, but that would make images rather hard to create. And, while it is true that the speed of drawing images is, in general, often a critical performance factor, the speed of copying images from system memory
to display memory is not particularly critical in Mode X. Important images can be stored in off-screen memory and copied to the screen via the latches much faster than even the speediest system memory-to-display memory copy routine could manage.

I'm not going to present a routine to perform Mode X copies from display memory to system memory, but such a routine would be a straightforward inverse of Listing 48.4.

**LISTING 48.4 L48-4.ASM**

; Mode X (320x240, 256 colors) system memory to display memory copy
; routine. Uses approach of changing the plane for each pixel copied;
; this is slower than copying all pixels in one plane, then all pixels
; in the next plane, and so on, but it is simpler; besides, images for
; which performance is critical should be stored in off-screen memory
; and copied to the screen via the latches. Copies up to but not
; including the column at SourceEndX and the row at SourceEndY. No
; clipping is performed. C near-callable as:
;
; void CopySystemToScreenX(int SourceStartX, int SourceStartY,
; int SourceEndX, int SourceEndY, int DestStartX,
; int DestStartY, char* SourcePtr, unsigned int DestPageBase,
; int SourceBitmapWidth, int DestBitmapWidth);

SC_INDEX equ 03c4h ;Sequence Controller Index register port
MAP_MASK equ 02h ; index in SC of Map Mask register
SCREEN_SEG equ 0a00h ;segment of display memory in Mode X

parms struc
SourceStartX dw 2 dup(?) ;pushed BP and return address
SourceStartY dw ? ;X coordinate of upper-left corner of source
SourceEndX dw ? ;X coordinate of upper-left corner of source
SourceEndY dw ? ;X coordinate of lower-right corner of source
; (the row at EndX is not copied)
DestStartX dw ? ;X coordinate of upper-left corner of dest
DestStartY dw ? ;Y coordinate of upper-left corner of dest
SourcePtr dw ? ;pointer in DS to start of bitmap in which
; source resides
DestPageBase dw ? ;base offset in display memory of page in
; which dest resides
SourceBitmapWidth dw ? ;# of pixels across source bitmap
DestBitmapWidth dw ? ;# of pixels across dest bitmap
; (must be a multiple of 4)

parms ends

RectWidth equ -2 ;local storage for width of rectangle
LeftMask equ -4 ;local storage for left rect edge plane mask

STACK_FRAME_SIZE equ 4

.model small
.code
public _CopySystemToScreenX
_CopySystemToScreenX proc near

push bp ;preserve caller's stack frame
mov bp,sp ;point to local stack frame
sub sp,STACK_FRAME_SIZE ;allocate space for local vars
push si ;preserve caller's register variables
cl
mov ax,SCREEN_SEG ;point ES to display memory
mov es,ax
mov ax,[bp+SourceBitmapWidth]
mul [bp+SourceStartY] ;top source rect scan line
add ax,[bp+SourceStartX]
add ax,[bp+SourcePtr] ;offset of first source rect pixel
mov si,ax ; in DS
mov ax,[bp+DestBitmapWidth]
shr ax,1 ;convert to width in addresses
shr ax,1
mov [bp+DestBitmapWidth],ax ;remember address width
mul [bp+DestStartY] ;top dest rect scan line
mov di,[bp+DestStartX]
mov cx,di
shr di,1 ;X/4 - offset of first dest rect pixel in scan line
shr di,1
add di,ax ;offset of first dest rect pixel in page
add di,[bp+DestPageBase] ;offset of first dest rect pixel
in ds
and cl,011b ;CL - first dest pixel's plane
mov al,11h ;upper nibble comes into play when plane wraps from 3 back to 0
shl al,cl ;set the bit for the first dest pixel's plane
mov [bp+LeftMask],al ;plane in each nibble to 1
mov cx,[bp+SourceEndX] ;calculate # of pixels across rect
sub cx,[bp+SourceStartX] ;rect
jle CopyDone ;skip if 0 or negative width
mov bx,[bp+RectWidth].cx
mov bx,[bp+SourceEndY]
sbb bx,[bp+SourceStartY] ;BX - height of rectangle
jle CopyDone ;skip if 0 or negative height
mov dx,SC_INDEX ;point to SC Index reg
mov al,MAP_MASK ;point SC Index reg to the Map Mask
out dx,al ;point DX to SC Data reg
CopyRowsLoop:
  mov ax,[bp+LeftMask] ;remember the start offset in the source
  mov cx,[bp+RectWidth] ;remember the start offset in the dest
CopyScanLineLoop:
  push si
  push di
  push bx
  mov al,1
  rol al,1 ;set mask for next pixel's plane
  cmc
  sbb bx,di,0 ;wrapping from plane 3 to plane 0
  loop CopyScanLineLoop
  pop bx
  add bx,[bp+DestBitmapWidth] ;point to the start of the next scan line of the dest
  pop si
  add si,[bp+SourceBitmapWidth] ;point to the start of the next scan line of the source
  dec bx ;count down scan lines
  jnz CopyRowsLoop

Chapter 48
Who Was that Masked Image Copier?

At this point, it’s getting to be time for us to take all the Mode X tools we’ve developed, together with one more tool—masked image copying—and the remaining unexplored feature of Mode X, page flipping, and build an animation application. I hope that when we’re done, you’ll agree with me that Mode X is the way to animate on the PC.

In truth, though, it matters less whether or not you think that Mode X is the best way to animate than whether or not your users think it’s the best way based on results; end users care only about results, not how you produced them. For my writing, you folks are the end users—and notice how remarkably little you care about how this book gets written and produced. You care that it turned up in the bookstore, and you care about the contents, but you sure as heck don’t care about how it got that far from a bin of tree pulp. When you’re a creator, the process matters. When you’re a buyer, results are everything. All important. *Sine qua non.* The whole enchilada.

If you catch my drift.