Dr. Bill's Notes on Mode-13 Graphics

©October, 2002

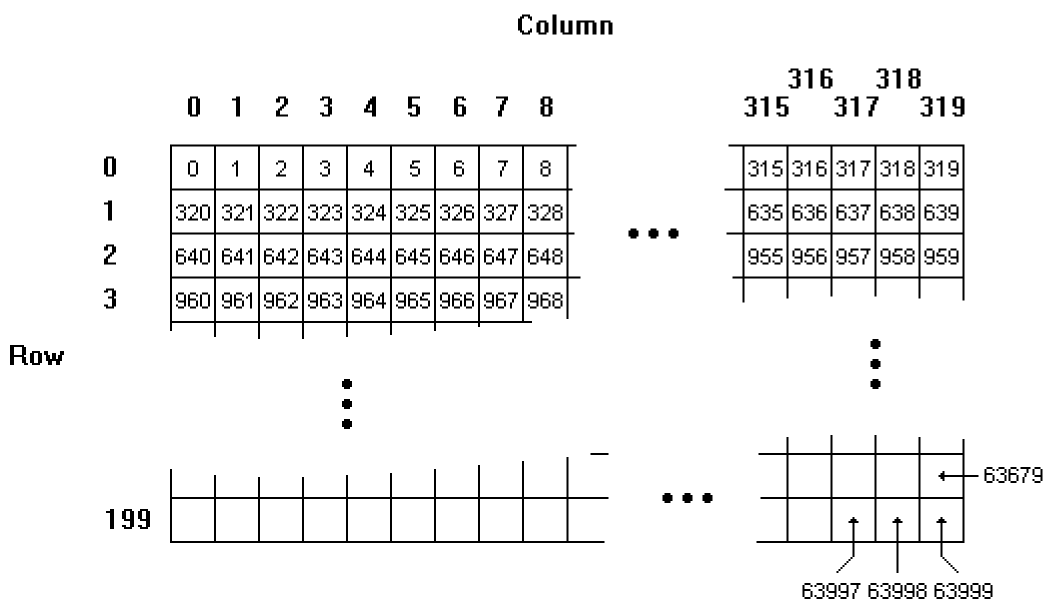
Dr. William T. Verts

Introduction

For beginning Intel 80x86 assembly language programmers, Mode-13 graphics provides a simple mechanism for exploring some of the important issues of any graphics system. It also is an excellent mechanism for understanding assembly language by developing more "interesting" programs than the typical toy programs of introductory computer science classes.

Mode-13 requires at least a VGA (or MCGA) graphics adapter. It is fairly coarse by today's standards, with images only 320 pixels wide by 200 pixels tall, and but 1 byte (256 colors) per pixel. A complete image is thus 320×200×1 bytes in size, or 64000 bytes. This is a particularly convenient size, as it is just under the 64K byte limit per segment on the earliest Intel x86 processors. Any larger, and the programmer would have to manage segment registers a lot more extensively.

Once the video mode has been set, the 64000 bytes are in a contiguous block starting at fixed memory location `A000:0000` (address `A0000`), and every pixel is a linear offset from that base address. Each raster line is exactly 320 bytes in length, so the addresses of two vertically adjacent pixels differ by exactly 320 bytes. The upper left pixel is at offset zero, as shown in the following image of the screen with byte offsets:



Entering Graphics Mode

Entering graphics mode is a simple call to the BIOS. In assembly language, that call is as follows:

```
MOV  AX,0013H
INT  10H
```

Interrupt `10H` is the BIOS call that controls the video system. The number `00H` (the value in `AH`) is the function number that corresponds to "Set the Video Mode", and the number `13H` (the value in `AL`) is the number of the desired mode.

Exiting Graphics Mode

Exiting graphics mode is as simple as entering graphics mode. In this case we wish to set the video mode to text-only, 25 lines by 80 columns of text. This is mode 3, and is accomplished by the following code:

```
MOV  AX,0003H
INT  10H
```

It is the same interrupt (`10H`) and function (`00H`) as before, but a different mode (`03H`).

Clearing the Screen

Clearing the screen means that the same pixel color is stored into all 64000 bytes of the video buffer area. If we assume that the pixel color is stored in memory at location `Color`, then the following code will step through the video screen one byte (pixel) at a time, storing the color value into each byte as it goes. The starting address of the video screen segment is put into the extra segment (`ES`) register, and base register `BX` starts at offset zero. The `CX` (counter register) is initialized with the number of steps to run a loop, and each pass through the loop stores one byte at `ES:BX`, increments `BX`, then decrements `CX` and repeats if `CX` is still nonzero. In the code fragment shown here, the comment fields represent a high-level pseudocode of the process being performed by the assembly language.

```
          MOV  AL,Color       ; AL gets the color value
          MOV  BX,0A000H      ;
          MOV  ES,BX          ; ES set to start of VGA
          MOV  BX,0           ; BX set to pixel offset 0
          MOV  CX,64000       ; CX set to number of pixels
                              ;
ClrLoop:                      ; Repeat
          MOV  [ES:BX],AL     ;    Memory[ES:BX] := Color
          INC  BX             ;    BX := BX + 1
          LOOP ClrLoop        ;    CX := CX - 1
                              ; Until CX = 0
```

There is a string instruction that makes this process more efficient. String instructions use special registers `SI` (source index) and `DI` (destination index) to step through the bytes of a string. The `STOSB` (Store-String-Byte) op-code can be interpreted as equivalent to `Memory[ES:DI] := AL` followed directly by `DI := DI + 1` in one instruction. Adding the `REP` prefix to the instruction repeats the instruction and decrements the `CX` (counter) register while `CX` is not zero. In order to use this one very fast instruction effectively, all of the prolog code must set up the special registers `AL`, `ES`, `CX`, and `DI` appropriately. (The `BX` register is not used, except in this example as a means of setting the `ES` register. In our example we could use `AX` instead, as long as setting the color into `AL` occurs afterwards.) The `REP STOSB` does the work of the entire While-loop to its right.

```
          MOV  AL,Color        ; AL gets the color value
          MOV  BX,0A000H       ;
          MOV  ES,BX           ; ES set to start of VGA
          MOV  CX,64000        ; CX set to number of pixels
          MOV  DI,0            ; DI set to pixel offset 0

     REP  STOSB                ; While CX <> 0 Do
                               ;    Memory[ES:DI] := AL
                               ;    DI := DI + 1
                               ;    CX := CX - 1
```

An even more efficient form uses `STOSW` (Store-String-Word). The `STOSB` example just shown stores one byte at a time into memory. This process can be sped up by nearly a factor of two by storing half as many two-byte words as there are individual bytes. Here the trick is to duplicate the pixel color value in both halves of the `AX` register so that storing a word sets two pixels at a time. Since there are half as many words as there are bytes, the number of times to loop can be cut in half from 64000 to 32000. The `STOSW` instruction increments `DI` by 2, and adding the `REP` prefix repeats the process while `CX` is not equal to zero.

```
          MOV  AL,Color        ; AL gets the color value
          MOV  AH,AL           ; Duplicate the color value
          MOV  BX,0A000H       ;
          MOV  ES,BX           ; ES set to start of VGA
          MOV  CX,32000        ; CX set to number of words
          MOV  DI,0            ; DI set to pixel offset 0

     REP  STOSW                ; While CX <> 0 Do
                               ;    Memory[ES:DI] := AX
                               ;    DI := DI + 2
                               ;    CX := CX - 1
```

This final form is the recommended method for setting all pixels in the screen to the same color. A variation of this technique will be examined again in the section on drawing horizontal lines.

Setting a Pixel

In order to set a pixel on screen, you need to know three things: the row address and column address of the pixel, and the pixel's new color. Valid row addresses are between 0 and 199, and valid column addresses are between 0 and 319. For these examples we are going to assume that the row address is stored in integer variable Y and the column address is stored in integer variable X, and that the values in those variables are in the appropriate legal ranges.

If the value of Y is less than 0 or greater than 199, or if the value of X is less than 0 or greater than 319, then the pixel routine should exit immediately. For legal values, the byte offset into the video area is determined by the computation:

```
Offset := Y * 320 + X
```

Thus, storing the new pixel color is implemented by the following pseudocode:

```
Memory[A000:Offset] := Color
```

In 80x86 assembly code, this same task is performed as follows:

```
MOV       AX,320          ;
IMUL      Y               ;
ADD       AX,X            ;
MOV       BX,AX           ; BX := Y * 320 + X

MOV       AX,0A000H       ; ES := A000
MOV       ES,AX           ; start of VGA area

MOV       AL,Color        ;
MOV       ES:[BX],AL      ; Store pixel byte
```

Of course, if the `ES` register is preloaded with `A000` and never changed, then this routine can be made faster by not including it again here.

To make a general-purpose subroutine, the values of X and Y should be passed to the routine on the stack instead of through memory. (The new pixel color can be passed through memory, since it is not likely to change frequently.) The calling sequence for such a subroutine is:

```
{compute X into AX}
PUSH      AX
{compute Y into AX}
PUSH      AX
CALL      Set_Pixel
```

Notice that the items pushed onto the stack before the call are not popped off after the subroutine returns. In 80x86 assembly language, the `RET` (return from subroutine) instruction can be modified to clear some number of bytes from the stack after the return address is popped. Since both X and Y are two-byte integers, the instruction `RET 4` will clear the parameters appropriately.

Inside the subroutine the values should be clipped to eliminate off-screen coordinates, and any registers used must be saved and restored to preserve their values outside of the subroutine. A complete such routine is as follows:

```
Set_Pixel PROC
          PUSH AX
          PUSH BX
          PUSH CX
          PUSH DX
          PUSH BP
          MOV  BP,SP

          MOV  AX,[BP+12]     ; Get Y from Stack
          CMP  AX,0           ;
          JL   EndPixel       ; Exit If Y<0
          CMP  AX,199         ;
          JG   EndPixel       ; Exit If Y>199

          MOV  BX,[BP+14]     ; Get X from Stack
          CMP  BX,0           ;
          JL   EndPixel       ; Exit If X<0
          CMP  BX,319         ;
          JG   EndPixel       ; Exit If X>319

          MOV  CX,320         ;
          IMUL CX             ; BX := Y*320+X
          ADD  BX,AX          ; (pixel offset)

          MOV  AX,0A000H      ;
          MOV  ES,AX          ; ES := VGA segment

          MOV  AL,Color       ;
          MOV  [ES:BX],AL     ; [ES:BX] := Color

EndPixel: POP  BP
          POP  DX
          POP  CX
          POP  BX
          POP  AX
          RET  4
Set_Pixel ENDP
```

While this routine is complete and effective, it is far too slow for many applications. In particular, drawing a horizontal line, where both end points are visible on screen, need not do *any* clipping and need only compute the byte offset for the first (leftmost) pixel of the line. This problem is addressed in the next section.

Drawing a Horizontal Line

Drawing a horizontal line is very similar to flooding the entire screen with a single color. The major difference is that the starting and ending pixels are within the same raster line, instead of being the first and last pixels of the entire graphics screen area. We can thus reuse a lot of the code for filling the screen (in particular the `STOSB` instruction) just as long as we set up the correct starting pixel address and the total number of pixels.

For a horizontal line we need to know the X coordinate of the left end of the line, the X coordinate of the right end of the line, and the Y coordinate of the line. Call these coordinates X1, X2, and Y. For now assume that X1 is less than or equal to X2, and assume that all three values are legal (i.e., point <X1,Y> and <X2,Y> are both on screen).

From this model we can calculate the offset into the graphics area of the starting pixel by the expression:

```
Offset := Y * 320 + X1
```

We can also calculate the number of pixels by the expression:

```
Pixels := X2 - X1 + 1
```

The 80x86 assembly language code to do this would be as follows:

```
     MOV  CX,X2          ;
     SUB  CX,X1          ;
     INC  CX             ; CX := Number of pixels

     MOV  AX,320         ; Compute pixel offset of
     MUL  Y              ; leftmost endpoint <X1,Y>
     ADD  X1             ;
     MOV  DI,AX          ; DI := Y * 320 + X1

     MOV  BX,0A000H      ;
     MOV  ES,BX          ; ES := VGA segment

     MOV  AL,Color       ; AL := color value

REP  STOSB               ; While CX <> 0 Do
                         ;    Memory[ES:DI] := AL
                         ;    DI := DI + 1
                         ;    CX := CX - 1
```

In order to make the horizontal line routine complete, we must first discard lines above or below the screen, insure that X1 is less than or equal to X2, and we must clip the ends of lines that extend beyond the left and right edges of the screen (possibly discarding the entire line in the process). In a high-level pseudocode, the process is as follows:

```
If Y  <   0 Then Exit ;
If Y  > 199 Then Exit ;
If X1 >  X2 Then Swap(X1, X2) ;
If X1 <   0 Then X1 := 0 ;
If X2 > 319 Then X2 := 319 ;
If X1 >  X2 Then Exit ;
{ Plot what remains of the line }
```

By clipping X1 against 0 and X2 against 319, lines that are entirely off the left side or right side of the screen will end up with X1 > X2 at the end of this process, so the last test discards those cases.

A general-purpose subroutine to draw horizontal lines requires a calling sequence similar to that of painting a single pixel on screen, except two X values must be pushed instead of one. This is as follows:

```
{compute X1 into AX}
PUSH      AX
{compute X2 into AX}
PUSH      AX
{compute Y into AX}
PUSH      AX
CALL      HLine
```

As with the paint pixel subroutine, the horizontal line subroutine will flush the parameters from the stack as it exits. The complete subroutine with clipping, written as a procedure, starts on the next page.

```
;-----------------------------------------;
; Stack at start of useful work:
;             X1        BP+18
;             X2        BP+16
;             Y         BP+14
;             RET ADR   BP+12
;             AX        BP+10
;             BX        BP+8
;             CX        BP+6
;             DX        BP+4
;             DI        BP+2
;    SP -->   BP        BP+0
;-----------------------------------------;

HLine         PROC NEAR
              PUSH AX
              PUSH BX
              PUSH CX
              PUSH DX
              PUSH DI
              PUSH BP
              MOV  BP,SP

              MOV  AX,[BP+14]     ; Get Y from Stack
              CMP  AX,0           ;
              JL   HLine_Done     ; Exit_If Y < 0
              CMP  AX,199         ;
              JG   HLine_Done     ; Exit_If Y > 199

              MOV  BX,320         ; Y := Y * 320
              IMUL BX             ;
              MOV  [BP+14],AX     ; Replace Y on Stack

              MOV  AX,[BP+18]     ; Get X1 from Stack
              MOV  BX,[BP+16]     ; Get X2 from Stack

              CMP  AX,BX          ; If X1 > X2 Then Swap(X1,X2)
              JLE  HLine_Sort     ;
              XCHG AX,BX          ;
    HLine_Sort:                   ;

              CMP  AX,0           ; If X1 < 0 Then X1 := 0
              JGE  DoneX1         ;
              MOV  AX,0           ;
     DoneX1:                      ;
```

```
            CMP  BX,319           ; If X2 > 319 Then X2 := 319
            JLE  DoneX2           ;
            MOV  BX,319           ;
     DoneX2:                      ;

            CMP  AX,BX            ; Exit_If X1 > X2 (clipped)
            JG   HLine_Done       ;

            SUB  BX,AX            ; CX := X2 - X1 + 1
            INC  BX               ;       (pixel count)
            MOV  CX,BX            ;

            ADD  AX,[BP+14]       ; DI := Y * 320 + X1
            MOV  DI,AX            ; (product is on stack
                                  ;  in Y's place)

            MOV  AX,0A000H        ; ES := VGA Segment
            MOV  ES,AX            ;

            MOV  AX,Color         ; AL := Color
            CLD                   ;
       REP  STOSB                 ; While CX > 0 Do
                                  ;    [ES:DI] := AL
                                  ;    DI := DI + 1
                                  ;    CX := CX - 1

    HLine_Done:
            POP  BP
            POP  DI
            POP  DX
            POP  CX
            POP  BX
            POP  AX
            RET  6
HLine       ENDP
```

Drawing a Vertical Line

Drawing vertical lines is not as simple as drawing horizontal lines because adjacent vertical pixels are not adjacent in memory, but instead are 320 bytes apart. Thus, the `STOSB` instruction cannot be used in this circumstance. We will have to step from one raster line to the next by adding 320 to an old address to get the new address.

A vertical line requires the X coordinate and two Y coordinates, called Y1, and Y2. As with the horizontal line, we are assuming that the point values <X,Y1> and <X,Y2> represent legitimate screen coordinates, and that Y1 is less than or equal to Y2. The starting pixel address is computed by the expression:

```
Offset := Y1 * 320 + X
```

and the number of pixels by the expression:

```
Pixels := Y2 - Y1 + 1
```

The 80x86 assembly language code for this process is then:

```
            MOV  CX,Y2          ;
            SUB  CX,Y1          ;
            INC  CX             ; CX := Number of pixels

            MOV  AX,320         ; Compute pixel offset of
            MUL  Y1             ; topmost endpoint <X,Y1>
            ADD  X              ;
            MOV  BX,AX          ; BX := Y1 * 320 + X

            MOV  AX,0A000H      ;
            MOV  ES,AX          ; ES := VGA segment

            MOV  AL,Color       ; AL := color value

VLine_Loop:                     ; Repeat
            MOV  [ES:BX],AL     ;    Memory[ES:BX] := AL
            ADD  BX,320         ;    BX := BX + 320
            LOOP VLine_Loop     ;    CX := CX – 1
                                ; Until CX = 0
```

Clipping is handled similar to that of the horizontal line routine: lines with X coordinate less than zero or greater than 319 are discarded, the Y1 and Y2 values are sorted, Y1 is clipped to 0, Y2 is clipped to 199, and whatever remains is plotted.

The calling sequence for a vertical line subroutine is as follows:

```
{compute X into AX}
PUSH      AX
{compute Y1 into AX}
PUSH      AX
{compute Y2 into AX}
PUSH      AX
CALL      VLine
```

The complete subroutine starts below:

```
;----------------------------------------;
; Stack at start of useful work:
;              X         BP+16
;              Y1        BP+14
;              Y2        BP+12
;              RET ADR   BP+10
;              AX        BP+8
;              BX        BP+6
;              CX        BP+4
;              DX        BP+2
;    SP -->    BP        BP+0
;----------------------------------------;

VLine          PROC NEAR
               PUSH AX
               PUSH BX
               PUSH CX
               PUSH DX
               PUSH BP
               MOV  BP,SP

               MOV  AX,[BP+16]     ; Get X from Stack
               CMP  AX,0           ;
               JL   VLine_Done     ; Exit_If X < 0
               CMP  AX,319         ;
               JG   VLine_Done     ; Exit_If X > 319

               MOV  AX,[BP+14]     ; Get Y1 from Stack
               MOV  BX,[BP+12]     ; Get Y2 from Stack

               CMP  AX,BX          ; If Y1 > Y2 Then Swap(Y1,Y2)
               JLE  VLine_Sort     ;
               XCHG AX,BX          ;
```

```
    VLine_Sort:                     ;

               CMP  AX,0            ; If Y1 < 0 Then Y1 := 0
               JGE  DoneY1          ;
               MOV  AX,0            ;
     DoneY1:                        ;

               CMP  BX,199          ; If Y2 > 199 Then Y2 := 199
               JLE  DoneY2          ;
               MOV  BX,199          ;
     DoneY2:                        ;

               CMP  AX,BX           ; Exit_If Y1 > Y2 (clipped)
               JG   VLine_Done      ;

               SUB  BX,AX           ; CX := Y2 - Y1 + 1
               INC  BX              ;       (pixel count)
               MOV  CX,BX           ;

               MOV  BX,320          ;
               IMUL BX              ; (Y1 still in AX)
               ADD  AX,[BP+16]      ;
               MOV  BX,AX           ; BX := Y1 * 320 + X

               MOV  AX,0A000H       ; ES := VGA Segment
               MOV  ES,AX           ;

               MOV  AL,Color        ; AL := Color

    VLine_Loop:                     ; Repeat
               MOV  [ES:BX],AL      ;    Memory[ES:BX] := AL
               ADD  BX,320          ;    BX := BX + 320
               LOOP VLine_Loop      ;    CX := CX - 1
                                    ; Until CX = 0

    VLine_Done:
               POP  BP
               POP  DX
               POP  CX
               POP  BX
               POP  AX
               RET  6
VLine          ENDP
```

General Lines

Plotting Circles

Saving and Plotting Image Regions