USING THE EGA FEATURE CONNECTOR FOR A 120-COLUMN DISPLAY

You've probably heard of people who replace the crystals in their PC ATs to make them run faster. Did you know you can do the same thing with your EGA? With a video adapter, however, faster doesn't mean faster. Instead, faster means more: more dots and a higher resolution. By using a faster crystal with your EGA, you can go beyond the normal 80-column display.

We'll show you how. For well under $15, you can construct the EGA External Clock Board described here that will let your EGA produce 120 columns on the Enhanced Color Display.

THE EGA DOT CLOCK: A BIT OF BACKGROUND In the EGA video modes that use 350 scan lines, the dot clock and all timing signals are derived from a 16.257-MHz crystal on the EGA board. Since your Enhanced Color Display requires a horizontal sync rate of 21.85 kHz, the EGA uses this 16.257-MHz dot clock to generate 744 dots per scan line, or 93 characters. Eighty of these characters are used in the display, 10 of them occur during the horizontal retrace, and the 3 left over are for the virtually nonexistent border.

If you replace the 16.257-MHz crystal with something faster, you can generate more dots—and thus more characters—while still maintaining the 21.85-kHz horizontal sync rate required by the monitor. With a 24-MHz crystal you can squeeze out 120 columns on the Enhanced Color Display. These extra columns, in combination with the 43-line mode, produce a total of 5,160 displayable characters instead of the meager 2,000 on the conventional PC display.

What is really amazing (for IBM) is that doing something like this is entirely supported by the EGA hardware and BIOS.

First, you don't need to actually replace the crystal on your EGA board with one of a higher frequency. The mysterious Feature Connector that is a part of every Enhanced Graphics Adapter can route an external dot clock to the EGA hardware. That's partly what the Feature Connector is designed for. Second, the EGA BIOS lets you set up an area in memory containing new video parameter values that the BIOS will load into the CRT Controller registers to provide the correct timings for the additional columns. Another register that the BIOS will load directs the EGA hardware to use the external crystal instead of the one on the board.

How high can this external dot clock frequency be? I've tested boards that use the Chips and Technologies EGA CHIP-Set with crystal oscillators up to 32 MHz. That makes a readable (but obviously tiny) 160-column display. The IBM EGA, however, starts having flicker problems when you go beyond 28 MHz, so you're limited to about 140 characters. EGA boards from the same manufacturer may also have different limits. Your own board may be limited to 26 MHz, or it could go even higher than 32 MHz.

But let's restrain ourselves. Once you have the EGA External Clock Board built, you can experiment to your heart's content. I'm going to stick to one example here that will use a 24-MHz crystal for 120 columns on the Enhanced Color Display. (A similar technique can be used to create 120 columns on the Monochrome Display, but it presents software complications that I'll discuss later.)

BUILDING THE FEATURE You can build the EGA External Clock Board with just four parts, three pieces of wire,

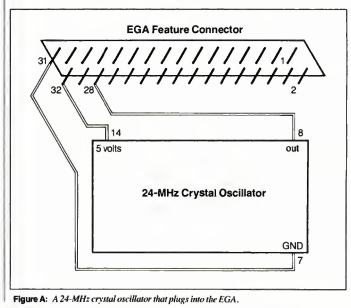

Figure A: *A 24-MHz crystal oscillator that plugs into the EGA.*

a soldering iron, and wire clippers.

The connection diagram is shown in Figure A. A 24-MHz crystal oscillator, the same size and shape as the 16.257-MHz crystal oscillator already on your EGA board, attaches to 3 pins on the EGA Feature Connector. Pins 32 and 31 provide 5 volts and ground, respectively, to power the oscillator. The 24-MHz output from the crystal oscillator connects to pin 28 of the Feature Connector.

Besides a little hook-up wire (less than a foot) and a little solder, all four of the parts you'll need can be purchased from JDR Microdevices, 1224 S. Bascom Ave., San Jose, CA 95128; (800) 538-5000 or (800) 662-6279 in Calif. The JDR prices and part numbers are shown in Figure B; equivalents are available from many electronic supply houses.

When you get the parts, first cut the

JDR Item No.	Description	Price
OSC24 0	24.0-MHz Xtal Oscillator	$4 95
HDR-80	2 × 40 header strip (breakable)	2 49
P25x45	Proto Board bare 2 5 × 4 5 inches	2 40
14 PIN ST	14-pin low-profile IC socket	11
		$9 95

Add $2.50 UPS ground or $3.50 UPS air delivery

Figure B: *A list of parts needed for the EGA External Clock Board.*

header strip (a knife will do it) so that you get a strip with 2 rows of 16 pins each. The long ends of the header strip will plug into the Feature Connector socket. Next, cut the perforated circuit board to about 1½ inches by 2 inches.

Figures C and D show the front and back, respectively, of the constructed board. When you install the header strip on the circuit board, make sure the short pins go through the holes in the board so that the long pins can plug into the Feature Connector. Push the pins of the DIP socket through the holes from the other side of the board. If you're making a wire-wrap version, you can glue the header strip and DIP socket to the board and wire-wrap the pins. Otherwise, you simply have to cut three pieces of wire

Figure C: *The front of the constructed board.*

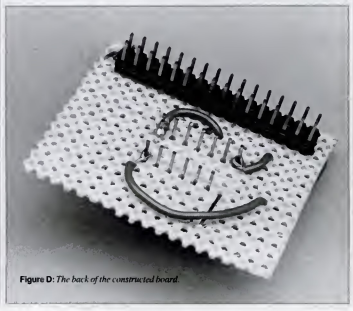

Figure D: *The back of the constructed board.*

("Using the EGA Feature Connector" continued)

and solder one end of each piece to the pins on the DIP socket and the other end to the long pins of the header strip. Note that the wires connect to the header strip pins on the front of the board and connect to the DIP socket pins on the hack of the board, so you'll have to run the wires from back to front through holes in the circuit board.

When everything is soldered and secure, clip any excess leads from the header and DIP socket to about ¼ inch. Plug the crystal oscillator into the DIP socket so that the text on the oscillator case is right side up when you hold the board with the header strip at the top. You may have to shorten the leads of the oscillator so it fits snugly into the socket.

Remove your EGA board from your PC and plug the EGA External Clock Board into the Feature Connector so that it is parallel to the EGA board. Put the EGA board back into the PC. Just be careful to make sure that the case of the crystal oscillator doesn't short out any contacts on the board adjacent to it. If it threatens to, just slip in an index card to separate the two boards.

SOFTWARE SUPPORT The EGA BIOS recognizes only four text modes (0 through 3) when the EGA is attached to an Enhanced Color Display. Although mode 0 is normally a 40-column mode that is not used by most programs, we can redefine mode 0 so it uses the external crystal oscillator instead of the on-board crystal oscillator. If we store new values to be loaded into the CRT Controller registers, the EGA will display 120 columns when it is switched to video mode 0.

Figure E lists a program called 120COLS that provides the required software support for the EGA External Clock Board. Use the 120COLS.SCR file with DEBUG to create 120COLS.COM, and then run 120COLS. You need only run this program once during your PC session, so you can put it in your AUTOEXEC.BAT file. Nothing will happen when you first run it. But now, try switching to

```
N 120COLS.COM
A 100
CLD
SUB     AX,AX
MOV     ES,AX
ES:
LDS     SI,[04A8]          ; Get SAV_PTR
ES:
MOV     WORD PTR [04A8],016C
ES:
MOV     [04AA],CS          ; Make it point here
PUSH    CS
POP     ES
MOV     DI,016C            ; Move addreeeee
MOV     CX,001C
REPZ
MOVSB
ES:
LDS     SI,[016C]          ; Get parameter ptr
MOV     DI,0100
MOV     CX,05C0
REPZ                       ; Copy parametere
MOVSB
PUSH    CS
POP     DS
MOV     WORD PTR [016C],0100
MOV     [016E],CS
MOV     SI,0149            ; Put new veluee in
MOV     DI,0640
MOV     CX,0023
REPZ
MOVSB
MOV     DX,0740            ; Remain resident
INT     27                 ; New veluee follow
DB      7B,18,0E
DW      1B00
DB      01,03,00,03
DB      AB
DB      07,77,7D,22,78,27
DB      6C,1F,00,0D,06,07
DB      00,00,00,00,5E,2B
DB      5D,3C,0F,5E,0A,A3
DB      FF

R CX
6C
W
Q
```

Figure E: *This 120COLS program makes video mode 0 a 120-column when using mode the external clock.*

video mode 0. You can do this either by executing

```
EGAMODE Ø
```

(which is a program shown in "Exploring the EGA, Part 1," *PC Magazine*, Volume 5 Number 14) or by executing

```
MODE BW4Ø
```

■ In a perfect world, programs would determine the current dimensions of the screen from the BIOS, but most just assume 80 columns.

with the regular PC-DOS MODE command. Your screen will now switch to a 120-column by 25-line display. To get back to normal, use

```
EGAMODE 3
```

or

```
MODE CO8Ø
```

While in 120-column mode, you can also run EGA43, shown in the accompanying article, to display 43 lines of 120 columns each.

You'll find that you can use DOS in 120-column mode, but pop-up utilities and applications programs may act oddly. In a perfect world, programs would determine the current dimensions of the screen from the BIOS and use those values, but you'll probably find that most programs just assume the screen is 80 columns wide. If these programs write to the screen through the BIOS, they will occupy only the left two-thirds of the display. If they write directly to the display memory, the beginning of the second row of text will start at column 81 of the first row on the screen and wrap around to the second row, and so forth.

This is too bad. When 120COLS has been loaded and video mode 0 is set, the EGA BIOS correctly sets up all the variables in the BIOS data area for other programs to use. Most PC programs, however, are "too smart" to use these variables.

The problem with doing something similar for an EGA connected to the Monochrome Display is this: The Monochrome Display has only one text mode, which is video mode 7. You won't want to redefine mode 7 to 120 columns, because then you will no longer have an 80-column monochrome mode. Any program that writes directly to the screen will almost certainly have problems with 120 columns.

WORDSTAR REVISITED In the accompanying article, however, I showed you how to patch *WordStar* to produce a 43-line display. You can also patch *WordStar* for 120 columns with or with-

```
N WS.COM
L
N WS120.COM
E 249 78
A 204
JNZ     201
MOV
JMP     300

A 201
MOV     AH,0F
INT     10
MOV     [120],AL
MOV     AX,4000
INT     10
RET

A 300
MOV     AL,[120]
DIB     AX,A0
INT     10
RET

W
Q
```

Figure F: *This script file creates WS120, which runs* WordStar *in 120-column mode.*

out a 43-line display. While this display isn't very useful for normal word processing, it's a real dream for program development in C or Pascal or any other languages that use nested loops.

If you plan to use 120 columns in DOS, you need only patch offset 249h in WS.COM to 78h to tell *WordStar* to use 120 columns.

Or you can use DOS as you normally would and have *WordStar* switch to 120-column mode. The script file shown in Figure F will create a new version of *WordStar* called WS120 that saves the video mode on entry and switches to video mode 0. On exit, it switches back to the original video mode. You can create this new version of *WordStar* simply by executing

```
DEBUG <WS120.SCR
```

You must run 120COLS during your PC session for this new version of *WordStar* to work right.

Figure G shows a similar script file that will create a *WordStar* program called WS43X120.COM that uses 43 lines of 120 columns. You can rename it to something shorter after you're done.

```
N WD.COM
L
N WS43X120.COM
E 249 2A
E 249 78
A 204
JNZ     201
JMP     300

A 201
MOV     AH,0F
INT     10
MOV     [120],AL
MOV     AX,0000
DIB     ES,AX
MOV     AX,1113
INT     10
MOV     CX,0907
MOV     AH,01
INT     10
RET

A 300
MOV     AL,[120]
DIB     AX,A0
INT     10
RET

W
Q
```

Figure G: *This script file creates WS43X120.COM, which uses a screen of 43 lines with 120 columns.*

FURTHER DIRECTIONS Since we've socketed the crystal oscillator on the EGA External Clock Board, you can try out different clock frequencies by substituting different crystal oscillators. To develop the corresponding software support, you'll need the EGA Technical Reference to learn about the organization of the video parameters and the programming of the CRT Controller registers. Use 120COLS as an example.

Readers familiar with digital logic circuitry might look at another aspect of the Feature Connector. Pins 20 and 21 are signals you can control from software using the Feature Control Register. You could put four different crystals and a 74153 multiplexer chip on an external board and select which crystal to use by programming this register.

If you have a NEC JC-1401P3A Multisync monitor (reviewed in "The EGA Standard: Monitors That Measure Up," *PC Magazine*, Volume 5 Number 6), you have even more options. The Multisync is not limited to a horizontal sync rate of 21.85 kHz, so you can use a higher crystal frequency to increase the number of scan lines as well as the number of dots per line. However, some code in the EGA BIOS, particularly that which recalculates EGA register values based on different font heights, always assumes 200 or 350 scan lines, so using more scan lines may not be easy.

In any event, it is clear that by using the Feature Connector to expand the capabilities of the EGA, the board becomes more powerful than anyone has yet suspected.—**Charles Petzold**

EGA has 128K, it can hold two fonts. A 64K EGA is limited to one font. Usually only one of these fonts is active at any one time, but you can tell the EGA which one you want to be active by using a simple BIOS call.

In addition, you can select any two of the four fonts to be active at the same time on the same screen. Instead of a normal 256-character set, this lets you use a 512-character set. Since the ASCII code for each character is still limited to 1 byte, something else outside the character code has to specify which font is to be used for each character.

This "something else" is the attribute bit that normally indicates foreground color intensity. Instead of (or in addition to) making a character high intensity, this bit selects the alternate font. Thus if your word processor can display all 256 normal characters and can also use high intensity to represent boldface on the screen, you can use boldface instead to select an additional 256 different characters on your screen.

■ An EGA with 256K can store four 256-character fonts in its memory. You can select two to be active at any one time.

You may have noticed that in the font-loading examples shown so far, register BL was always set to zero. Register BL specifies what the EGA BIOS calls the "Block to Load." This register can be set to any value from 0 through 3, and it corresponds to the four fonts you can load into an EGA equipped with 256K. If the value of BL is set to 1, the font is loaded into block 1 instead of block 0. Since block 0 is normally used for both low-intensity and high-intensity characters, you won't see a change in your display until you instruct the EGA to use the intensity bit to select the alternative font.

An example will make this clearer. Let's go back to EGAITAL (Figure 1),

■ **PROGRAMMING/UTILITIES**

which loads an italic character set. EGAITAL can actually take a parameter that specifies the block (0 through 3) into which to load the font. If no parameter is specified, it loads the font into block 0. But now try running

```
EGAITAL 1
```

Earlier, when we ran EGAITAL with no parameter, the text on the screen became italic. Now, nothing happens. The italic font has been loaded into the EGA memory, but in block 1 instead of block 0. Block 0 still contains the normal font.

Now we have to tell the EGA two things: first, that we want block 0 to be used for low-intensity characters and block 1 to be used for high intensity; second, that we want to suppress the normal use of the intensity bit in the attribute byte.

Figure 11 shows a program called EGA512 that will do this. It takes two parameters that may range from 0 through 3. The first parameter indicates the font to use on low-intensity characters; the second is the font to use for high-intensity characters. If you execute

```
EGA512 0 1
```

anything that is high intensity on your screen will become low-intensity italic. Anything that is low intensity will remain the same.

To prove this, execute TEST512, shown in Figure 12. This program prints two lines, one in low intensity and one in high intensity. The high-intensity line will not be high intensity, but, rather, will be italic. Now, if you execute

```
N EGA512.COB
A 100
MOV     BL,[80581]      ; 1st Parameter
AND     BL,03           ; Lower 3 bits
MOV     BH,[80601]      ; 2nd Parameter
AND     BH,03           ; Lower 3 bits
MOV     CX,04           ; Save in CX
SHL     BH,1            ; Shift 4H left
SHL     BH,1
SHL     BH,1
SHL     BH,1
OR      BL,BH           ; Combine with BL
MOV     AX,1103         ; Set block spec.
INT     10
MOV     BX,0F12         ; Enable all planes
CMP     CL,CH           ;   but only if
JE      0134            ;   blocks are equal
MOV     BH,00           ; Otherwise low
MOV     AX,1000         ; Enable planes
INT     10
INT     20

N CX
20
W
Q
```

Figure 11. *EGA512 lets you use a 512-character set based on the foreground intensity bit of the attribute.*

```
N TEST512.COM
A 100
MOV     AH,0F   ; Get video page
INT     10
MOV     AH,03   ; Get cursor
INT     10
MOV     DX,0128 ; String 1
MOV     CX,000F ; Number of chars
MOV     BL,07   ; Attribute
MOV     AX,1301 ; Write String
INT     10
MOV     AH,03   ; Get cursor
INT     10
MOV     DX,0137 ; String 2
MOV     CX,0018 ; Number of chars
MOV     BL,07   ; Attribute
MOV     AX,1301 ; Write String
INT     10
INT     10
DB      "Low Intensity",0D,0A
DB      "High Intensity",0D,0A

N CX
47
W
Q
```

Figure 12. *TEST512 prints two strings in low and high intensity to test or demonstrate a 512-character set.*

```
EGA512 1 0
```

everything in low intensity will be italics (the block 1 font), and everything in high intensity will be the normal font (block 0).

Finally,

```
EGA512 0 0
```

turns everything back to normal. It makes both low intensity and high intensity have the same font. The EGA512 program checks if the two parameters are the same and, if so, restores the meaning of the attribute bit to specify high intensity.

If your word processor works in text mode, does on-screen display of boldface, and doesn't reset the video mode on entry, try executing

```
EGA512 0 1
```

and see if you can get italic in your word processor instead of boldface. (Of course, then you have the problem of printing those extra 256 characters on your printer. But this is an article about video. Your printer is a whole other problem.)

You may be distressed at losing a high-intensity character display when you use the 512-character set, but there's a way around that, too. In part 1, I showed you how to use the EGA palette registers to map the 4-bit IRGB code into a 6-bit rgbRGB code. With this technique you can map the low-intensity IRGB codes (where the "I" intensity bit is 0) into high-intensity rgbRGB codes. The only thing you're really limited to with a 512-character set is 8 foreground colors instead of 16 for the two fonts, since only the RGB bits of the attribute (and not the I bit) are used for foreground intensity. Using the palette registers, you can map these into anything you want.

Another alternative to restoring high-intensity foreground is to invert all the bits of the two fonts and use another BIOS call (AX = 1003h and BL = 0) to enable background intensity instead of blinking. Now, since your characters are really background, you have all 16 colors available for the two fonts. Your background is now limited to 8 colors (which is normal), and you can't use hardware blinking.

The underline on the monochrome display presents another problem. If you decide to remap palette registers to give you low and high intensity for each font, you'll be able to use an underline only in either low or high intensity, but not both.

THE POOR MAN'S FONT EDITOR

Now that you've seen how you can load fonts into the EGA, you may want to create your own fonts. There are two ways to do it. You could start off with a piece of graph paper, draw each character by coloring in the appropriate blocks, translate each row of dots into a two-digit hexadecimal number, and then type these numbers into a file. This is the long way.

■ To create your own fonts, we'll be turning your text editor into a font editor.

It's easier to use an on-screen "font editor" program. You can then begin with an existing font and make modifications to it. Of course, font editors can be complex, but we'll be taking an inexpensive approach by turning your own text editor into a font editor. You'll be editing an assembly language file that contains a series of assembler "DB" statements. You can then convert this file back into bytes with the IBM or Microsoft Macro Assembler.

Figure 13 shows a program called

■ PROGRAMMING/UTILITIES

FONT2DB that retrieves an existing EGA font and translates it into these DB statements. The file created by this program is quite long: there are 256 characters and 14 bytes per character (for the 8 by 14 font), and each byte uses one line. The bytes are shown in binary format as 0s and 1s, so you get a visual representation of the font on your screen. Although FONT2DB is set up to list the 8 by 14 font, I've also shown

```
N FONT2DB.COM
A 100
MOV     BH,02     ; 03 for 8x8 Font!
MOV     AX,1130
INT     10        ; ES:BP points to font
MOV     CX,0100   ; Number of characters
MOV     BH,0E     ; 08 for 8x8 Font!
ES:
MOV     DH,[BP+00]        ; Get Byte
INC     BP                ; Point to next
MOV     BL,08             ; 8 bits
MOV     DI,013C           ; Destination
MOV     BYTE PTR [DI],00; Zero it out
RCL     DH,1              ; Get the bit
ADC     BYTE PTR [DI],30; Put in 0 or 1
INC     DI                ; Next bit
DEC     BL                ; Bit counter
JNZ     0116              ; Loop around
MOV     DX,0130           ; Point to output
MOV     AH,09             ; And print it
INT     21
DEC     BH        ; Bytes per character
JNZ     010C      ; Loop around
MOV     DX,0145   ; Print out return
MOV     AH,09
INT     21
LOOP    010A      ; Get next character
INT     20
DB      "DB 01234567b",0D,0A,"$"

R CX
48
W
Q
```

Figure 13. *FONT2DB prints an EGA font in binary format so that you can use your text editor to customize your own fonts.*

in the .SCR file which values to change to use it for the 8 by 8 font.

If you simply execute

```
FONT2DB
```

you'll get a listing of the font on your screen. By redirecting standard output,

```
FONT2DB >MYFONT
```

the font will be saved in a file called MYFONT. Now you can load MYFONT into your text editor, turn off insert mode, and make changes to the font by replacing 0s with 1s where you want a dot, and 1s with 0s where you don't want a dot.

To change the font height, simply delete or add lines. You can use from 1 to 32 lines per character, but make sure each character has the same number of bytes, or your font will look quite strange.

After you've created your own font, the next step requires the IBM or Microsoft Macro Assembler. A shell program called FONTLOAD.ASM is shown in Figure 14. The INCLUDE statement includes your edited font in the assembly. For more versatility, you may want to assemble your fonts independently and rewrite the loader program to load them from disk files.

INTRODUCING EGA GRAPHICS

So far, in both parts of this article, I've shown you EGA features that you can use in text modes. (And you thought you bought your EGA for better graphics!)

Programming graphics on the EGA is no picnic. If you are interested in creating graphics applications for the EGA, you should seriously consider using a "virtual device interface," such as that found in *Microsoft Windows* or IBM's *Graphics Development Toolkit*. The *Graphics Development Toolkit*, for instance, contains routines that can be accessed from IBM Pascal, IBM FORTRAN, IBM BASIC compiler, or Lattice C (but not the BASIC interpreter). Also, both the BASICA interpreter in DOS, Version 3.2, and Microsoft QuickBASIC, Version 2.0, support EGA graphics.

If you have lots of time to while away waiting for your screen to be painted, you could also use the BIOS for graphics. The EGA BIOS supports the same Read Dot and Write Dot function calls supported by the system board BIOS for the CGA. The only difference is that register BH must contain the video page. The BIOS Write Dot call is extremely slow, about 0.5 milliseconds per dot on a normal 4.77-MHz PC or XT, so it takes a little under 2 minutes to refresh an entire 640 by 350 screen in the simplest case. Figure 15 shows the program (called WRITEDOT) that I used to

```
; FONTLOAD.ASM -- Loads a customized font

CSEG    Segment
        Assume  CS:CSEG, DS:CSEG, ES:CSEG
        Org     0100h
Entry:  Mov     AX, 1100h          ; Load font
        Mov     BH, 0Eh            ; Bytes/Char
        Mov     BL, 00h            ; Block
        Mov     CX, 0100h          ; Number Chars
        Mov     DX, 0000h          ; Start Char
        Mov     BP, Offset Font ; Address
        Int     10h
        Int     20h

Font    Label   Byte

Include MYFONT  ; Your font file name goes here

CSEG    EndS
        End     Entry
```

Figure 14. *FONTLOAD loads a customized font in the format produced by FONT2DB. This program requires a Macro Assembler.*

time this. You can see for yourself how bad it is.

If none of these methods is good enough and you want to program your own graphics, you'll be writing directly to the video memory and programming the EGA registers. There is no better way to learn EGA graphics than to dig into the EGA Technical Reference and sweat it out. I'll show you some basics to get you started.

You'll get the fastest graphics routines by programming in assembly language. You can also program EGA graphics routines directly in any high-level language that supports BIOS calls, register output commands, and direct access of memory in a far segment.

Probably the best high-level language for programming EGA graphics is C, and that is what I've used in the EGADEMO program shown in Figure 16. This program assumes your EGA is connected to an Enhanced Color Display. EGADEMO sets the video mode to 16, displays 100 random rectangles on your screen, sets the video mode to 16 again, and displays another 100 random rectangles by XORing (eXclusive ORing with the colors already on the screen). It then sets the video mode back to 3.

```
N WRITEDOT.COM
A 100
MOV     AX,0010 ; Set video mode
INT     10      ;   to 16
SUB     BH,BH   ; Page 0
SUB     DX,DX   ; Row 0
SUB     CX,CX   ; Column 0
MOV     AL,01   ; Blue
MOV     AH,0C   ; Write Dot
INT     10
INC     CX      ; Next Column
CMP     CX,0280 ; See if 640
JB      010B    ; If below, loop
INC     DX      ; Next Row
CMP     DX,015E ; See if 350
JB      0109    ; If below, loop
MOV     AX,0003 ; Set video mode
INT     10      ;   to 3
INT     20      ; Terminate

R CX
26
W
Q
```

Figure 15. *WRITEDOT demonstrates that the speed of the BIOS Write Dot routine makes it unsuitable for most graphics applications.*

EGADEMO was written for Microsoft C, Version 3. Library functions that correspond to `int86` and `outp` may be different for other C compilers, and so also may be the use of a far pointer for directly accessing the screen.

With Microsoft C, Version 3.00, the program can be compiled thus:

```
MSC EGADEMO /Ze;
LINK EGADEMO;
```

The /Ze parameter is necessary to enable the `bold` far keyword.

GRAPHICS MEMORY ORGANIZATION When you program graphics on the EGA, you'll be writing directly to the screen memory. This memory is organized differently from the memory in CGA graphics modes. So let's first review what we know about the CGA and see how the EGA is different.

In the CGA-compatible graphics modes (4 through 6), the display memory occupies 16K, beginning at segment address B800h. In medium-resolution modes (4 and 5), the 320 dots of each scan line require 80 bytes. Each byte represents 4 pixels with 2 bits each for the four colors. In high-resolution black-and-white mode 6, the 640 horizontal dots also require 80 bytes, each byte representing 8 pixels. In either case, 16,000 bytes are required for the 200 scan lines. The only complication is that all the even scan lines are at the top of this memory, followed by all the odd scan lines.

The display memory in EGA graphics modes 13 through 16 occupies the 64K segment beginning at segment address A000h. In graphics mode 16 (350 scan lines with 640 dots each and 16 colors), one screen's worth of data requires 80 bytes per scan line times 350 scan lines times 4 (for the 16 colors), or 112,000 bytes. Two video pages are available, bringing total memory to 224,000—just below the maximum 256K on the EGA.

This 256K of display memory fits into a 64K memory space by being organized into four 64K color planes. The four color planes represent the blue, red, green, and intensity bits that define the 16 colors. Each of the four color planes is addressed the same way. You specify which color plane you want to write to or read from by using the EGA registers accessible through output ports.

The byte at address A000:0000 is the leftmost 8 bits in the first scan line of the display. The byte at address A000:0001 is the next 8 bits in the top scan line. For graphics modes 14 through 16, the byte at address A000:0050 is the first 8 bits in the second scan line. For graphics mode 13, the second scan line begins at address A000:0028. The most significant bit always represents the leftmost dot.

Although the division of EGA graphics memory into identically addressed color planes initially appears clumsy, it has some distinct advantages. Since you can write to all four color planes simultaneously, you can actually write up to 32 bits of

```
/* EGADEMO.C -- Compile with Microsoft C Version 3 with /Ze parameter
                to enable "far" keyword.  Programmed by Cherilee Petzold. */

#include <dos.h>
#define MAXROWS 350
#define MAXCOLS 640
#define COLBYTES MAXCOLS/8
#define GRAFOUT(index,value) { outp(0x3CE,index) ; outp(0x3CF,value) ; }
#define SWAP(a,b,temp) { temp = a ; a = b ; b = temp ; }

main()  {
        setvideomode(16) ;
        randomrect(8) ;                         /* Drew colored rectangles */
        setvideomode(16) ;
        randomrect(0x18) ;                      /* Rectangles with XORing */
        setvideomode(3) ;
        }

setvideomode(mode)                              /* Sets video mode */
        int mode ;
        {
        union REGS regs ;

        regs.x.ax = mode ;
        int86(0x10, &regs, &regs) ;
        }

randomrect(fnct)                                /* Draws 100 rectangles */
        int fnct ;
        {
        unsigned int i, r1, c1, r2, c2, temp ;

        GRAFOUT(3, fnct) ;                      /* Function Register */
        for (i = 0 ; i < 100 ; i++) {
                r1 = rand() % MAXROWS ; r2 = rand() % MAXROWS ;
                c1 = rand() % MAXCOLS ; c2 = rand() % MAXCOLS ;
                if (r1 > r2) SWAP (r1, r2, temp) ;
                if (c1 > c2) SWAP (c1, c2, temp) ;
                rect(r1, c1, r2, c2, rand()%16);
                }
        }

rect(r1, c1, r2, c2, color)                     /* r1 <= r2 and c1 <= c2 */
        unsigned int r1, c1, r2, c2, color ;
        {
        char far *addr = (char far *) (0xA0000000L + COLBYTES*r1 + (c1>>3)) ;
        int numrows    = r2 - r1 ;              /* Number of rows */
        int numcols    = (c2 >> 3) - (c1 >> 3) - 1 ;    /* Number of columns
*/
        char lmask     =  0xFF >> (c1 & 7) ;    /* Mask for left */
        char rmask     = ~(0xFF >> (c2 & 7)) ;  /* Mask for right */
        register row, col ;

        if (numcols < 0) {                      /* Adjustment for small rows
*/
                lmask &= rmask ;
                rmask = numcols = 0 ;
                }
        GRAFOUT(0, color) ;                     /* Set/Reset Register */
        GRAFOUT(1, 0xFF) ;                      /* Enable Set/Reset Register
*/
        for (row = 0 ; row < numrows ; row++) {
                GRAFOUT(8, lmask) ;                     /* Bit Mask Register
*/
                *(addr++) &= 1 ;                        /* Left side */
                GRAFOUT(8, 0xFF) ;
                for (col = 0 ; col < numcols ; col++)
                        *(addr++) &= 1 ;                /* Middle part */
                GRAFOUT(8, rmask) ;
                *(addr++) &= 1 ;                        /* Right side */
                addr += COLBYTES - 2 - numcols ;        /* Next row */
                }
        GRAFOUT(0, 0) ;                         /* Set/Reset to normal */
        GRAFOUT(1, 0) ;                         /* E/R Enable to normal */
        GRAFOUT(8, 0xFF) ;                      /* Bit mask to normal */
        }
```

Figure 16. *EGADEMO is an EGA graphics demonstration program written in C that draws random rectangles on the screen.*

data (8 bits of four colors) by writing 1 byte to the display memory. Partially for this reason, the *wrong* way to program EGA graphics is to start off with a simple "Write Dot" routine and base all other graphics routines on this one function. You'll get much better speed if you try to take advantage of the EGA's memory organization rather than work against it.

HOW EGADEMO WORKS EGADEMO illustrates the use of four of the EGA's graphics registers. Programming these registers correctly is necessary before you write graphics information to the display memory. These use the GRAFOUT macro. In each case an index value is first written to output port 3CE. This defines which register we want to write to. Next, the value we want to write to that register is written to output port 3CF. Other EGA registers have different port addresses and work differently.

The first use of these registers is in the `randomrect`. The value written to the Function Register (with index equal to 3) tells the EGA whether any data written to the display will be written without modification or will be ANDed, ORed, or XORed with the data already there. EGADEMO draws 100 rectangles without modification and 100 more with XORing. You can also program this register to rotate data being written to the display.

The `rect` function programs the Set/Reset Register with the color of the rectangle and also programs the Set/Reset Enable Register to enable the set/reset function for all four color bits. This is not the only way to handle color on the EGA, but I've found it to be the most convenient for solid fills.

The `rect` function also programs the Bit Mask Register. Since 1 byte of the EGA memory encompasses 8 horizontal pixels, you often need to protect some pixels from modification. Any bit in the Bit Mask Register set to 0 protects that bit from modification when you write to the display if that memory location has just been read.

Let me elaborate. When you read a byte at a particular address from EGA memory, the contents of all four color planes at that address are latched internally in the EGA hardware. This is a total of 32 bits, 8 in each color plane. When you write a byte at an address, either the contents of these latches are written back to memory if the corresponding Bit Mask Register bit is 0, or something else is if the bit is 1. In the `rect` function, the data written to each bit where the Bit Mask Register is set to 1 is the color value in the Set/Reset Register.

The `rect` function uses the Bit Mask Register for the left and right sides of the rectangle. This allows for rectangles that may not start or end on a byte boundary, since we want to protect the data to the left and right of the rectangle.

The C statement

```
*(addr++) &= 1
```

appears three times in the `rect` function: for the left side of the rectangle, the central part, and the right side. This statement looks as if it's ANDing the contents of the video memory with 1, but it's really not. All this statement has to do (besides increment the address when it's done) is read a byte to latch the internal EGA registers and then write a byte to write the Set/Reset color value to the unprotected bits. The value we actually read or write out makes no difference. All we have to do is perform a read and a write. This statement happens to do it very quickly.

(Watch out for your compiler, however. I originally had a 0 at the right of the expression shown above, but Microsoft's C compiler "optimized" the statement to write only a 0 to memory! That would be OK in most cases, but not for the EGA.)

As I mentioned earlier, using the EGA Set/Reset Register is not the only way to do this, but it does indicate some of the strangeness involved in programming graphics for the EGA. The Write Dot routine in the EGA BIOS uses a different (but no simpler) approach involving the Map Mask Register. Having these two examples in front of you will help when you start exploring the EGA registers.

But as we've seen, there is much in the EGA to explore without getting anywhere near graphics. While the EGA will help make the transition to graphics-based software more pleasant to us, it opens up more potential in character modes as well.

Charles Petzold is a contributing editor of PC Magazine.