Inside the EGA

by
Michael Abrash

This is the first of a series of articles about graphics. Actually, it's about more than just graphics—it's about algorithms and assembly-language programming and all manner of tricks for putting IBM PC-family microcomputers through their paces. Still, the focus will be on graphics of the EGA/CGA/Hercules sort, all the while keeping an eye on higher-performance boards that have the potential to become standards someday soon. If there's any particular area you'd like to see covered or interesting findings you'd like to share, let me know.

I'm going to start with a look at the Enhanced Graphics Adapter (EGA). The CGA and Hercules boards are pretty simple, and a great deal has been written about how to program them. The EGA, on the other hand, is a difficult board to program efficiently (the more so because IBM's documentation is hard to obtain and harder to understand), and little information about programming the EGA has seen print. The EGA's tremendous potential—far greater than that of the CGA—makes it well worth knowing about, and that's what the next few articles will be about.

The EGA

Although the EGA is fast becoming the standard graphics adapter in the IBM PC marketplace, it has not met with universal approval. The knock against the EGA is that its performance is inadequate. True, the EGA is not an ultra-high-resolution board, but it offers a solid price/performance ratio. Sixteen colors at a resolution of 640 x 350, complete with a monitor and decent compatibility with older software, for under $800 is a pretty good deal—that's about what I paid for 320 x 200 in 4 colors just four years ago. (I'm talking about EGA clones, of course—the IBM EGA and monitor are greatly overpriced.) In my opinion, the EGA provides adequate resolution and color for the majority of PC applications at a price most users can afford.

There's another facet of graphics performance, though, and that's the speed at which programs can update the screen. Bit-mapped graphics tend to be slow when handled by the main processor: Witness Microsoft Windows, which really requires an AT for decent speed, or, for that matter, the Macintosh, which expends much of the power of an 8-MHz 68000 in supporting a bit-mapped interface. The time spent in bit-map manipulation goes up rapidly as the size of the bit map and the complexity of controlling it increase. The EGA's bit map is 16 times as large as the CGA's, and although the EGA has many useful features built in, those features are controlled through dozens of registers. The result: It's hard to produce fast graphics on the EGA.

Hard, but not impossible—and that's why I like this odd board. It's a throwback to an earlier generation of micros, when inventive coding and a solid understanding of the hardware were the best tools for improving performance. Increasingly, faster processors and powerful coprocessors are seen as the solution to the sluggish software produced by high-level languages and layers of interfacing and driver code, and that's surely a valid approach. However, there are hundreds of thousands of EGAs installed right now, with no coprocessors to help and hardly an 80386 in sight. What's more, because the EGA is an 8-bit device, and because of display memory wait states, an AT isn't as much a help as you'd expect. The upshot is that only a good low-level coder who understands the EGA can push the board to its potential.

I'll be exploring the EGA by selecting a specific algorithm or feature and implementing code to support it on the EGA, examining aspects of the EGA architecture as they become relevant. You'll get to see EGA features in context, where they are more comprehensible than in IBM's somewhat arcane documentation, and you'll get working code to use or to modify to meet your needs. If there's a particular aspect of the EGA that you'd like to read about, send me a letter and I'll try to work it in.

The prime directive of EGA programming is that there's rarely just one way to program the EGA for a given purpose. Once you understand the tools the EGA provides, you'll be able to combine them to generate the particular synergy your application needs. My EGA routines are not intended to be taken as gospel, or to show "best" implementations, but rather to start you down the road to understanding the EGA. Let's begin.

An Introduction to the EGA

Most discussions of the EGA start out with a traditional "Here's a block diagram of the EGA" approach, with lists of registers and statistics. I'll get to that eventually, but you can find it in IBM's EGA documentation and several other magazines. Besides, it's numbing to read specifications and explanations, and the EGA is an exciting board. It's the sort of board that makes you want to get your hands dirty under the hood, to write some nifty code just to see what the board can do. What's more, the best way to understand the EGA is to see it work, so let's jump right into a sample of the EGA in action, getting a feel for the EGA's architecture in the process.

Listing 1 is a sample EGA program that pans around an animated 16-color high-resolution (640 x 350) playfield. There's a lot packed into this code; I'm going to take advantage of the technically high level of *PJ* readership and assume you can figure out the non-EGA aspects for yourself. I'm not going to explain how the ball is animated, for example. (For an introduction to animation, see "Animation Techniques for the IBM PC," by Michael Abrash and Dan Illowsky, *PC Tech Journal*, July 1986.) What I will do is cover each of the EGA features used in this program—virtual screen, vertical and horizontal panning, color plane manipulation, multi-plane block copying, and page flipping—at a conceptual level, letting the code itself demonstrate the implementation details.

Some Background

A little background is necessary before we can begin to examine Listing 1. The EGA is built around four VLSI (Very Large Scale Integrated) chips, named the CRT Controller (CRTC), the Timing Sequencer (TS), the Attribute Controller (ATC), and the Graphics Data Controller (GDC). There are two GDCs per EGA since each GDC controls two of the EGA's four planes of memory. Some EGA compatible chip sets combine several of these chips into a single chip or shuffle specific functions among chips, but the programming interface always looks the same since otherwise the chip set would not be EGA-compatible.

Each of these chips has a sizeable complement of registers. It is not particularly important that you understand why a given chip has a given register; all the registers together make up the programming interface, and it is the entire interface that is of interest to the EGA programmer. However, the means by which most EGA registers are addressed makes it necessary for you to remember which registers are in which chips.

Most EGA registers are addressed as internally indexed registers. The internal address of the register is written to a given chip's index register, and then the data for that register is written to the chip's data register. For example, GDC register 8, the Bit Mask register, is set to 0FFh by writing 8 to port 3CEh, the GDC Index register, and then writing 0FFh to port 3CFh, the GDC Data register. Internal indexing makes it possible to address the 9 GDC registers through only two ports and allows the entire EGA programming interface to be squeezed into less than a dozen ports. The downside is that two I/O operations are required to access most EGA registers.

The ports used to control the EGA are shown in the table below. The CRTC, TS, and GDC Index registers are located at the addresses of their respective Data registers plus one. However, the ATC Index and Data registers are located at the same address, 3C0h. The function of this port toggles on every OUT to 3C0h and resets on every read from the Input Status 1 register (3DAh when the EGA is in color modes, 3BAh in monochrome modes). Note that all CRTC registers are addressed at either 3DXh or 3BXh, the former in color modes and the latter in monochrome modes. This provides compatibility with the register addressing of the Color/Graphics and Monochrome Adapters.

The ports through which the EGA is controlled

These are the official addresses, but some EGA ports are incompletely decoded and can be addressed at several ports. Most importantly, the ATC Index/Data register can be addressed at 3C1h as well as at 3C0h.

Register	Address
ATC Index/Data register	3C0h (write only)
Input Status 0 register	3C2h (read only)
TS Index register	3C4h (write only) ✓
TS Data register	3C5h (write only) ✓
Graphics 2 Position register	3CAh (write only)
Graphics 1 Position register	3CCh (write only)
GDC Index register	3CEh (write only)
GDC Data register	3CFh (write only)
CRTC Index register	3B4h/3D4h (write only) ✓
CRTC Data register	3B5h/3D5h (write only) ✓
Input Status 1 register/ ATC Index/Data reset	3BAh/3DAh (read only)
Feature Control	3BAh/3DAh (write only)

The method used in the EGA BIOS to set registers is to point DX to the desired index register, load AL with the index, perform a byte OUT, increment DX to point to the data register (except in the case of the ATC, where DX remains the same), load AL with the desired data, and perform a byte OUT. A handy shortcut is to point DX to the desired index register, load AL with the index, load AH with the data, and perform a word OUT. Since the high byte of the OUT value goes to port DX + 1, this is equivalent to the first method but is considerably faster. This works for even the ATC since the ATC Index and Data registers are decoded at both 3C0h and 3C1h; however, be sure that the ATC Index/Data register is set to index mode before programming the ATC with a word OUT (you can ensure this by first reading the Input Status 1 register to reset ATC addressing).

How safe is this method of addressing EGA registers? I have run into accelerator boards that had trouble with word OUTs; however, all such problems I am aware of have been fixed. Moreover, Microsoft Windows uses word OUTs, even to the ATC, so any clone computer or EGA that doesn't support word OUTs could scarcely be considered a clone at all.

A speed tip: The setting of each chip's index register remains the same until it is reprogrammed. This means that in cases where you are setting the same internal register repeatedly, you can set the index register to point to that internal register once, then write to the data register multiple times. For example, the Bit Mask register (GDC register 8) is set repeatedly inside a loop when drawing lines. The standard code for this is

```
MOV     DX,03CEH   ;point to GDC Index register
MOV     AL,8       ;internal index of Bit Mask register
OUT     DX,AX      ;AH contains Bit Mask register setting
```

Alternatively, the GDC Index register could initially be set to point to the Bit Mask register with

```
MOV     DX,03CEH   ;point to GDC Index register
MOV     AL,8       ;internal index of Bit Mask register
OUT     DX,AL      ;set GDC Index register
INC     DX         ;point to GDC Data register
```

and then the Bit Mask register could be set repeatedly with the byte-size OUT instruction:

```
OUT     DX,AL      ;AL contains Bit Mask register setting
```

which is four cycles faster than a word-sized OUT, and which does not require AH to be set, freeing up a register. Be aware that this method works only if the GDC Index register remains unchanged throughout the loop.

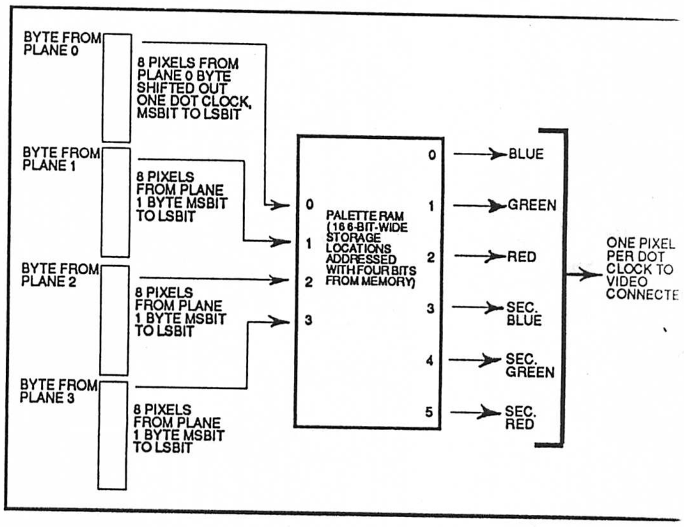
Linear Planes and True EGA Modes

The EGA's memory is organized as four 64K planes. (The base model from IBM comes with only 16K per plane, but this configuration doesn't support the 16-color high-resolution mode. So far as I can tell, this version of the EGA is about as widely used as the PC's cassette port, so I will discuss only 256K EGA operation.) Each of these planes is a linear bit map; that is, each byte from a given plane controls eight adjacent pixels on the screen, the next byte controls the next eight pixels, and so on to the end of the scan line. The next byte then controls the first eight pixels of the next scan line, and so on to the end of the screen. This will no doubt come as a relief to programmers weary of compensating for the CGA's two-bank architecture and the Hercules Graphics Card's four-bank architecture.

The EGA adds a powerful twist to linear addressing; the logical width of the screen in EGA memory need not be the same as the physical width of the display. The programmer is free to define all or part of the EGA's large memory map as a logical screen of up to 4096 pixels in width and then use the physical screen as a window onto any part of the logical screen. What's more, a virtual screen can have any logical height up to the capacity of EGA memory. Such a virtual screen could be used to store a spreadsheet or a CAD/CAM drawing, for instance. As we will see shortly, the EGA provides excellent hardware for moving around the virtual screen; taken together, the virtual screen and the EGA's smooth panning capabilities can generate very impressive effects.

All four linear planes are addressed in the same 64K memory space starting at A000:0000. Consequently, there are four bytes at any given address in EGA memory. The EGA provides special hardware to assist the CPU in manipulating all four planes, in parallel, with a single memory access so that the programmer doesn't have to spend a great deal of time switching between planes. Astute use of this EGA hardware allows EGA software to equal the performance of CGA software even though the EGA bit map is much larger than the CGA bit map.

Each memory plane provides one bit of data for each pixel. The bits for a given pixel from each of the four planes are combined into a nibble that serves as an address into the EGA's palette RAM, which maps the one of sixteen colors selected by display memory into any one of sixty-four colors, as shown in Figure 1. All sixty-four mappings for all sixteen colors are independently programmable.



BYTE FROM PLANE 0
8 PIXELS FROM PLANE 0 BYTE SHIFTED OUT ONE DOT CLOCK, MSBIT TO LSBIT
BYTE FROM PLANE 1
8 PIXELS FROM PLANE 1 BYTE MSBIT TO LSBIT
BYTE FROM PLANE 2
8 PIXELS FROM PLANE 1 BYTE MSBIT TO LSBIT
BYTE FROM PLANE 3
8 PIXELS FROM PLANE 1 BYTE MSBIT TO LSBIT
0
1
2
3
PALETTE RAM (16 6-BIT-WIDE STORAGE LOCATIONS ADDRESSED WITH FOUR BITS FROM MEMORY)
0
1
2
3
4
5
BLUE
GREEN
RED
SEC. BLUE
SEC. GREEN
SEC. RED
ONE PIXEL PER DOT CLOCK TO VIDEO CONNECTE

Figure 1. Video data from EGA display memory to video connector

The EGA BIOS supports several graphics modes (modes 4, 5, and 6) in which EGA memory appears not to be organized as four linear planes. These modes exist for CGA compatibility only and are not true EGA graphics modes. Use them when you need CGA-type operation and ignore them the rest of the time. The EGA's special features are most powerful in true EGA modes, and it is on these modes (modes 0Dh, 0Eh, 0Fh, and 10h) that I will concentrate. EGA text modes, which feature soft fonts, are another matter entirely, to be explored separately at a later date.

With that background out of the way, we can get on to the sample EGA program shown in Listing 1. I suggest you run the program before continuing since the explanations will mean far more to you if you've seen the features in action.

Smooth Panning

The first thing you'll notice upon running the sample program is the remarkable smoothness with which the display pans from side-to-side and up-and-down. That the display can pan at all is made possible by two EGA features: the 256K bit map and the virtual screen capability. Even the most memory-hungry of the EGA modes, mode 10h, uses only 28K per plane, for a total of 112K out of the total 256K of EGA memory. Consequently, there is room in EGA memory to store more than two full screens of video data. In the sample program, memory is organized as two virtual screens, each with a resolution of 672 x 384, as shown in Figure 2.

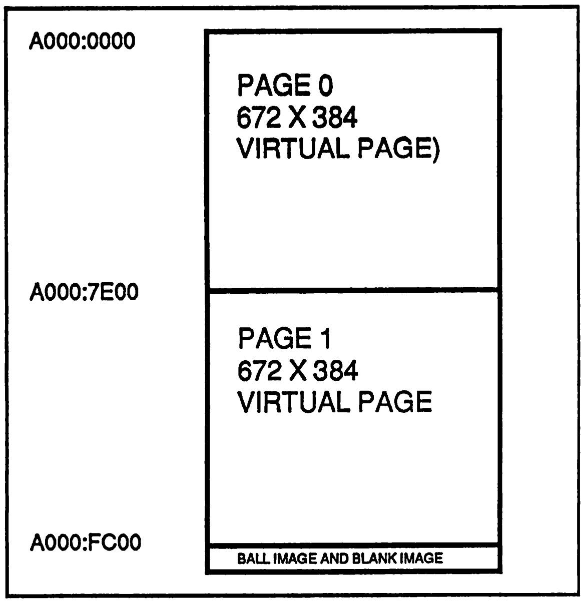


Figure 2. Organization of EGA memory in sample program. Space is reserved for two 672 x 384 virtual pages and for images of the ball and the blank.

The area of the virtual screen actually displayed at any given time is selected by setting the display memory address at which to begin fetching video data; this is set by way of the start address registers (Start Address High, CRTC register 0Ch, and Start Address Low, CRTC register 0Dh). Together these registers make up a 16-bit display memory address at which the CRTC begins fetching data at the beginning of each video frame. Increasing the start address causes higher-memory areas of the virtual screen to be displayed. For example, the Start Address High register could be set to 80h and the Start Address Low register could be set to 00h in order to cause the display screen to reflect memory starting at offset 8000h in each plane, rather than at the default offset of 0.

The logical height of the virtual screen is defined by the amount of EGA memory available. As the EGA scans display memory for video data, it progresses from the start address toward higher memory one scan line at a time until the frame is completed. Consequently, if the start address is increased, lines farther towards the bottom of the virtual screen are displayed; in effect, the virtual screen appears to scroll up on the physical screen.

The logical width of the virtual screen is defined by the Offset register (CRTC register 13h), which allows redefinition of the number of words of display memory considered to make up one scan line. Normally, 40 words of display memory constitute a scan line; after the CRTC scans these 40 words for 640 pixels' worth of data, it advances 40 words from the start of that scan line to find the start of the next scan line in memory. This means that displayed scan lines are contiguous in memory. However, the Offset register can be set so that scan lines are logically wider (or narrower, for that matter) than their displayed width. The sample program sets the Offset register to 2Ah, making the logical width of the virtual screen 42 words, or 42 * 2 * 8 = 672 pixels, as contrasted with the actual width of the hi-res screen, 40 words or 640 pixels. The logical height of the virtual screen in the sample program is 384; this is accomplished simply by reserving 84 * 384 contiguous bytes of EGA memory for the virtual screen, where 84 is the virtual screen width in bytes and 384 is the virtual screen height in scan lines.

The start address is the key to panning around the virtual screen. The start-address registers select the row of the virtual screen that maps to the top of the display; panning down a scan line requires only that the start address be increased by the logical scan line width in bytes, which is equal to the Offset register times two. The start address registers select the column that maps to the left edge of the display as well, allowing horizontal panning, although in this case only relatively coarse byte-sized adjustments—panning by eight pixels at a time—are supported. Smooth horizontal panning is provided by the Horizontal Pel Panning register, ATC register 13h, working in conjunction with the start address. Up to 7 pixels' worth of single pixel panning of the displayed image to the left is performed by increasing the Horizontal Pel Panning register from 0 to 7. This exhausts the range of motion possible via the Horizontal Pel Panning register; the next pixel's worth of smooth panning is accomplished by incrementing the start address by one and resetting the Horizontal Pel Panning register to 0. Smooth horizontal panning should be viewed as a series of fine adjustments in the 8-pixel range between coarse byte-sized adjustments.

A horizontal panning oddity: Alone among EGA modes, monochrome EGA text mode has 9 dots per character clock. Smooth panning in this mode requires cycling the Horizontal Pel Panning register through the values 8, 0, 1, 2, 3, 4, 5, 6, and 7. 8 is the "no panning" setting.

There is one annoying quirk about programming the ATC. When the ATC Index register is set, only the lower five bits are used as the internal index. The next most significant bit, bit 5, controls the source of the video data sent to the monitor by the EGA. When bit 5 is set to 1, the output of the palette RAM, derived from display memory, controls the displayed pixels; this is normal operation. When bit 5 is 0, video data does not come from the palette RAM, and the screen becomes a solid color. The only time bit 5 of the ATC Index register should be 0 is during the setting of a palette RAM register since the CPU is able to write to palette RAM only when bit 5 is low. Immediately after the palette RAM is set, however, 20h should be written to the ATC Index register to restore normal video, and at all other times bit 5 should be set to 1.

By the way, palette RAM can be set via the BIOS video interrupt (interrupt 10h), function 10h. Whenever an EGA function can be performed reasonably well through a BIOS function, as it can in the case of setting palette RAM, it should be, both because there is no point in reinventing the wheel and because the BIOS may well mask incompatibilities between the IBM EGA and EGA clones. For instance, it is not easy to set all the palette RAM registers without causing momentary flicker, and the exact parameters for doing this vary somewhat among EGA clones. However, it's a safe bet that the palette RAM-setting routines in every BIOS are flicker-free.

Color Plane Manipulation

The EGA provides a powerful array of hardware assistance for manipulating the four display memory planes. Two features illustrated by the sample program are the ability to control which planes are written to by a CPU write and the ability to copy four bytes—one from each plane—with a single CPU read and a single CPU write.

The Map Mask register (TS register 2) selects which planes are written to by CPU writes. If bit 0 of the Map Mask register is 1, each byte written by the CPU will be written to EGA memory plane 0, the plane that provides the video data for the least significant bit of the palette RAM address. If bit 0 of the Map Mask register is 0, CPU writes will not affect plane 0. Bits 1, 2, and 3 of the Map Mask register similarly control CPU access to planes 1, 2, and 3, respectively. Any of the sixteen possible combinations of enabled and disabled planes can be selected. Beware, however, of writing to an area of memory that is not zeroed. Planes that are disabled by the Map Mask register are not altered by CPU writes, so old and new images can mix on the screen, producing unwanted color effects as, say, three planes from the old image mix with one plane from the new image. The sample program solves this by ensuring that the memory written to is zeroed. A better way to clear memory is provided by the set/reset capabilities of the EGA, which I'll cover another time.

The sample program writes the image of the colored ball to EGA memory by enabling one plane at a time and writing the image of the ball for that plane. Each image is written to the same EGA addresses; only the destination plane, selected by the Map Mask register, is different. You might think of the ball's image as consisting of four colored overlays, which together make up a multicolored image. The sample program writes a blank image to EGA memory by enabling all planes and writing a block of zero bytes; the zero bytes are written to all four EGA planes simultaneously.

The images are written to a nondisplayed portion of EGA memory in order to take advantage of a useful EGA hardware feature, the ability to copy all four planes at once. As shown above, four times as many reads and writes—and several OUTs as well—are required to copy a multicolored image into EGA memory as would be needed to draw the same image in the CGA's high-resolution mode. This causes unacceptably slow performance, all the more so because the wait states that occur on accesses to EGA memory make it very desirable to minimize display memory accesses.

The solution is to take advantage of the EGA's write mode 1, which is selected via bits 0 and 1 of the GDC Mode register (GDC register 5). (Be careful to set bits 2-7 properly for the current display mode when setting bits 0 and 1.) In write mode 1, a single CPU read loads the addressed byte from all four planes into the EGA's four internal latches, and a single CPU write writes the contents of the latches to the four planes. During the write, the byte written by the CPU is irrelevant.

The sample program uses write mode 1 to copy the images that were previously drawn to the high end of EGA memory into a desired area of display memory, all in a single block copy operation. This is an excellent way to keep the number of reads, writes, and OUTs required to manipulate the EGA's display memory low enough to allow real-time drawing.

The Map Mask register can still mask out planes in write mode 1. All four planes are copied in the sample program because the Map Mask register is still 0Fh from when the blank image was created.

The animated images appear to move somewhat jerkily because they are byte-aligned and so must move a minimum of 8 pixels horizontally. This is easily solved by storing rotated versions of all images in EGA memory and then in each instance drawing the correct rotation for the pixel alignment at which the image is to be drawn.

Page Flipping

CGA graphics typically flicker and/or ripple, an unavoidable result of modifying display memory at the same time that it is being scanned for video data. The extra display memory of the EGA makes it possible to perform page flipping, which eliminates such problems. The basic premise of page flipping is that one area of display memory is displayed while another is being modified. The modifications never affect an area of memory as it is providing video data, so no undesirable side effects occur. Once the modification is complete, the modified buffer is selected for display, causing the screen to change in a single frame. The other buffer is then available for modification.

Graphics mode page flipping is not possible with the CGA because the CGA's display memory is barely large enough to hold a single screen's bit map (a single page). However, the EGA has 64K per plane, enough to hold two pages and more even in hi-res mode. (A hi-res page is 80 times 350, or 28,000 bytes.) For page flipping, two non-overlapping areas of display memory are needed. The sample program uses two 672 x 384 virtual pages, each 32,256 bytes long, one starting at A000:0000 and the other starting at A000:7E00. Flipping between the pages is as simple as setting the start address registers to point to one display area or the other.

The timing of the switch between pages is critical to achieving flicker-free animation. It is critical that the program never be modifying an area of display memory as that memory is providing video data. This means that the start address should be changed at some time during the frame after which the page flip is to occur since the start address for a given frame is latched from the start address registers at the end of the previous frame. The most readily available vertical display status on the EGA is the vertical sync pulse status, available at bit 3 of the Input Status 1 register (addressed at 3BAh/3DAh). A logical approach to page flipping would seem to be to wait for the leading edge of the vertical sync pulse, then set the start address registers.

Unfortunately, the vertical sync bit is not the ideal status to monitor for the end of the frame since it doesn't begin until part way through the vertical retrace period, after the start address has been latched into the linear address counting circuitry. If the sample program were to draw to the supposedly undisplayed page during the frame after the start address is changed to the other page, flicker would occur; at this time, the new start address has not yet taken effect, so that page is actually still being displayed. To avoid this, the sample program waits twice for the leading edge of the vertical sync pulse, once to be sure about when the starting address registers are being set and once to allow the page flip to become effective.

Waiting for the sync pulse has the side effect of causing program execution to synchronize to the EGA's frame rate of 60 per second. In a program where all drawing can be done during a single frame time, this synchronization has the useful consequence of causing the program to execute at the same speed on an AT as on a PC.

As mentioned above, the vertical sync bit is not a perfect status to monitor for the end of a frame. There is a way to detect the true beginning of the EGA's vertical retrace period, via the vertical interrupt; for a sample implementation, see "Software Sprites for the IBM EGA and CGA" by Michael Abrash and Dan Illowsky, *PC Tech Journal*, August 1986.

An important point illustrated by the sample program is that while the EGA's bit map is far larger and more versatile than is the case with other adapters, it is nonetheless a limited resource and must be used judiciously. The sample program uses EGA memory to store two 672 x 384 virtual pages, leaving only 1024 bytes free to store images. In this case, the only images needed are a colored ball and a blank block with which to erase it, so there is no problem, but many applications require dozens or hundreds of images. The tradeoffs between virtual page size, page flipping, and image storage must always be kept in mind when designing programs for the EGA.

Just an Introduction

That pretty well covers the important points of the sample EGA program in Listing 1. To see the program run in 640 x 200 16-color mode, comment out the HIRES_VIDEO_MODE equate; in this mode, the display screen is much smaller relative to the virtual screen, so the smooth panning is far more pronounced. There are many EGA features we haven't even touched on, but the object is to give you a feel for the variety of features available on the EGA, to convey the flexibility and complexity of the EGA's resources, and to point out how different from the CGA the EGA is. Starting with the next article, I'll begin to systematically explore the EGA on a more detailed basis.

EGA Clones

There are an amazing number of EGA clones on the market now. Most are built around the Chips and Technologies chip set, but several other chip sets are in use as well. The EGA clones sell for so much less than the IBM version that it's hard to imagine that anyone but the most conservative corporate purchaser would buy the real thing, particularly since all the clones seem to be more than adequately compatible with the IBM EGA.

The clones are not exactly like the IBM EGA, however. Some have bugs in the chip set implementation, others have slightly different board layouts, and all have BIOS ROMs that differ to some extent from IBM's. Some manufacturers have also made a conscious decision to deviate slightly from the IBM standard in order to build a better board; for example, several manufacturers have provided modes with higher resolution than the EGA can muster.

What this means to the programmer is that EGA code should be tested on several EGA clones before being released. Given the tremendous popularity of the Chips and Technology EGA-clone chip set, it is not enough to design for the IBM EGA; I suspect that there are more C & T-based EGAs out there than IBM EGAs. In truth, the EGA standard is less standard than, say, the PC standard, because the PC is built out of off-the-shelf hardware, not custom VLSI chips, and because the EGA video BIOS is twice as large as the entire PC BIOS. There's no solution to this, but broad testing is a reasonable substitute.

The Macro Assembler

The code I'll be presenting will generally be written in Assembler. I think C is a good development environment, but I'm in general agreement with Hal Hardenbergh's assertion that the best code (although not necessarily the easiest to write or the most reliable) is written in Assembler. This is especially true of graphics code for the 8086 family, given segments and the string instructions, and for real-time programming of a complex board like the EGA, there's really no other choice.

Before I'm deluged with protests from C devotees, let me add that the majority of my productive work is done in C; no programmer is immune to the laws of time, and C is simply a faster environment in which to develop, particularly when working in a programming team. I'd like to emphasize that there are many excellent C compilers available for the PC, some of which produce remarkably good code, and all of which are a damn sight more reliable and easier to develop with than the Microsoft Macro Assembler. In fact, I think MASM is a serious impediment to assembly-language programming on a computer that is inherently hard to program in Assembler. My experience with MASM began with version 1.0 compiling a DB that had a DUP factor of 0 into a block of 64 apparently random bytes, and matters have continued in pretty much the same vein ever since. Version 4.0 is much faster and has fewer bugs, but it's far from perfect. For one thing, macros don't

E

REAL - TIME
GRAPHICS
ANIMATION

For the IBM PC, XT, AT or compatible, with Color Graphics Adapter and color monitor.

With this plug-in accessory, your PC can produce graphics as fast as expensive dedicated graphics work stations. The card acts as a graphics coprocessor to the CPU. Graphics primitives are drawn by an HD63484 chip. No need to compute the adresses of individual pixels in a line, the card takes the ends of the line as input then draws the line at high speed. The line appears on the screen at a million color pixels per second.

Keep your color graphics adapter and standard RGBI monitor, this card uses both. Since your old adapter remains a part of the system, you are sure that your machine remains completely IBM compatible. Gives you all 16 colors on the screen at the same time, even with 640x200 dots.

Are you tired of having to program graphics applications in assembler and C? This card comes with an include file for Turbo Pascal. Nearly 70 procedures and functions allow you to use a high level language to write fast graphics programs. In addition to the usual lines, circles, ellipses, rectangles, 2-color area fills, etc., this package includes support for real-time animation. You can swap in a hidden screen during vertical retrace. Your animation will be clean and synchronized - no hash the easy way. Automatic checkerboard dithering is supported, giving 256 psuelo colors on a standard IBM color monitor.

Envious of your friends with Amigas and STs? This card can redraw a 21x21 3-D net graph in 1/60 sec. That is 50400 vectors per second. And the card includes support for 3-D figure drawing and rotation.

It even works with your speed up cards and fast mother-boards. No memory addresses are used, and only four bytes of I/O ports are needed, so the board won't conflict with your other graphics adapters or peripherals.

This card is available only directly from the factory. Two cards are available, one with a resolution of 640x200, the other one goes to 640x400 but requires a long persistence phosphor monitor. Both cards come with a 30 day money-back trial period. Try the card, if it doesn't perform as claimed, or if you are unsatisfied with it for any reason send it back for a full refund. The cards also come with a 180 day warranty, and include free software updates.

EHGCp2	640x200	only	$374.00	Check or Money Order 30 day trial
EHGCp4	640x400	interlace	578.00	180 day warranty (714) 261-1725
	Shipping and Handling		4.00	

E-Heart Engineering / 18103 Sky Park South #D / Irvine, CA 92714

Number 67 on the Reader Service Card

always work the way they should, and it can be very difficult just to pin down the source of an error in a nested macro, let alone devise a workaround. For another, the whole structure of the assembly language is, to put it charitably, odd. I've never seen another strongly-typed assembler before, and I hope I never do. I've also never seen another assembler capable of generating a phase error, surely one of the most maddening and least useful error messages ever. I use STRUC and GROUP and OFFSET, and I know a slew of tricks like using LEA or DGROUP:OFFSET to get an accurate offset in a group, but is all of this really necessary? In my opinion, MASM is somewhere between a true assembler and a high-level language, with most of the disadvantages of both.

One example, and then I'll get to the point. When a structure element is used in C, the compiler knows which structure definition that element is associated with and consequently has no trouble with elements of different structures that have the same names. For instance, LinkedListBlock.Status and DescriptorBlock.Status can coexist happily in the same module. Not so in Assembler—structure element names seem to simply translate into symbols, just as if an EQU had been performed, and the second structure element with a given name produces a redefinition error. As a result, I find myself naming elements like LLBStatus and DBStatus, a far cry from the power of C structures.

So what's the point? Well, I'm willing to bet that someone out there knows a way around the STRUC problem I've just described. Perhaps it's not a problem at all, and I've simply misunderstood the Macro Assembler manual, or perhaps there's a good workaround. Whether this particular problem is solvable or not, there are many tricks being used in similar situations to wring utility out of MASM. Like it or not, MASM is pretty much it for assembly-language development for the PC family. True, there are other assemblers, but all my existing code and most of the tools and high-level languages I use are designed to work with MASM, and every time I try to leave MASM I run into incompatibilities that send me running back to the old standard. Since MASM is so important, I'll be writing about the MASM tricks I know, and I'd appreciate it if you'd write in and share your MASM experiences with *PJ*'s readers as well.

Two unsolved MASM puzzles to start things off:

(1) I've never figured out how to use STRUC to define negative offsets from BP in a stack frame. C routinely addresses passed parameters at positive offsets from BP and dynamic storage at negative offsets, but since STRUC seems to generate only positive offsets, I've had to put dynamic storage at positive offsets and set BP only after dynamic storage has been allocated. Although this works, it effectively halves the number of bytes that can be addressed with single-byte offsets from BP, potentially increasing code size and execution time.

(2) While developing ROMable code, I needed to force certain code to be at a certain offset (say offset 2000h) in the final COM file. My thought was to use something like

```
        DB      (2000h-$) DUP 0
```

to force the code to assemble at the desired location. Alas, the assembler can handle only a constant in this context, and "$" is not a constant. I tried a variety of other approaches, such as ORGing to the offset, but couldn't get the assembler to produce the desired result. I ended up disassembling the file, calculating the needed offset, and hardwiring it into a DB, which was fine until the next time I changed the code and forgot to adjust the DUP value in the DB. Does anyone know how to force code to assemble at an absolute location?

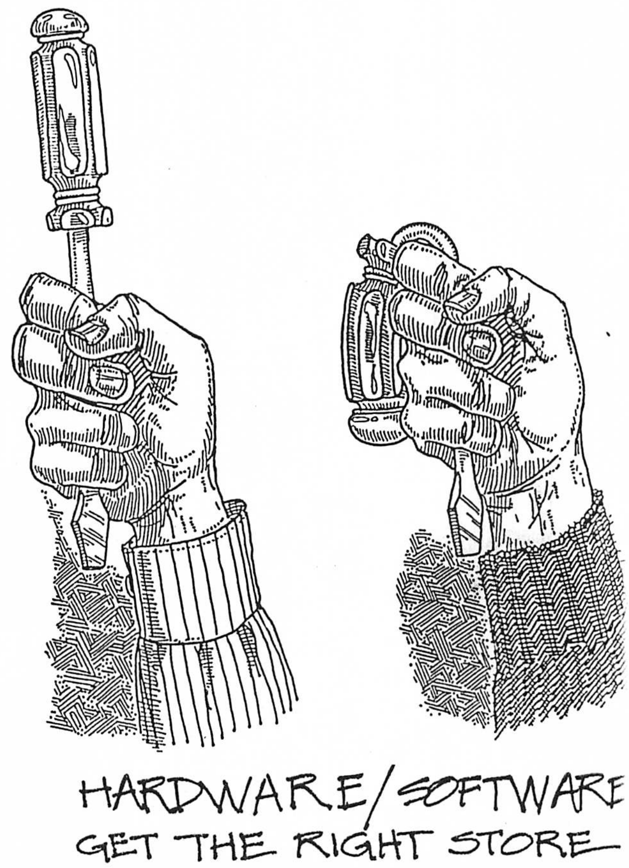
So. If you know the answers, let us know. If you've got bugs and workarounds of your own, let us know. We MASM developers can use all the help we can get.

Coming Up

Next time I'll look into the hardware assistance the EGA provides the CPU during display memory access. There are four latches and four ALUs in those chips, along with some useful masks and comparators; it should be interesting. See you then. □

Michael Abrash is a Senior Software Engineer for Orion Industries of Redwood City, CA, a manufacturer of PC-based instrumentation and microprocessor development systems.

Code follows.



Parallel Processing with the EGA

by
Michael Abrash

This issue's title refers to the ability of the powerful EGA chip set to manipulate up to four bytes of display memory at once. The EGA provides four ALUs (Arithmetic Logic Units) to assist the CPU during display memory writes, and this hardware is a tremendous resource in the task of manipulating the EGA's massive bit map. The ALUs are actually only one part of the surprisingly complex data flow architecture of the EGA, but since they're involved in almost all memory access operations, they're a good place to begin.

There's more to cover, too–starting with this article, I'm going to briefly discuss interesting graphics hardware and software I've come across recently. Since most *PJ* readers are developers, I'm not going to be doing traditional reviews, but rather discussing products' strengths and weaknesses from the perspective of a developer who's deciding what products to support and/or develop for. This issue, rather than examine specific products, I'm going to report on the state of PC graphics, as observed at Fall Comdex.

Comdex

The overriding graphics-related impression from Comdex was that the EGA is the standard for PC graphics. Virtually no one showed a CGA-level product, and just about all demos were run on EGAs or more powerful boards. One AT-compatible portable was shown: It had an EGA built into the motherboard. Many more built-in EGAs will surely follow, especially given the one- and two-chip EGA chip sets hitting the market. In short, imperfect as it may be, the EGA is the current standard for mass-market PC graphics. (On a related note, inexpensive ATs abounded at Comdex. The widespread use of ATs will help greatly in overcoming the relatively sluggish performance of the EGA.)

Given that the EGA is today's standard, the big question is clearly, "What's next?" One answer seems to be what many people are calling the EEGA, which is an EGA with higher-resolution modes than the IBM board's, typically 640 x 480 but ranging in some cases as high as 800 pixels in width and 500 pixels in height. Several EEGA boards were shown at Comdex, and others have already been announced. There's nothing revolutionary about the EEGA; it's a natural outgrowth of the development of EGA chip sets that can run faster than IBM's. EEGA hi-res modes are identical to the normal EGA 640 x 350 mode, with 16 of 64 colors available, except that the dot clock speed is higher and the portion of display memory used for video data at any given time is larger. (In 640 x 480 mode, for example, the active bit map size is over 37K; by contrast, 640 x 350 mode requires less than 28K.)

EEGAs are significant for several reasons. First, they provide enough resolution for emulation of many popular graphics terminals. Second, the market is being flooded with monitors that support the higher resolutions of these boards, such as the NEC MultiSync and Sony MultiScan. Third, 640 x 480 provides about a 1-to-1 aspect ratio; a 1-to-1 aspect ratio is one reason why Macintosh graphics look so good. Finally, support of EEGAs requires very little new programming since all the registers and memory addresses that are used in standard EGA programming are used by EEGAs as well, so software should become available for them quickly.

EEGAs are hardly a giant step forward, but they are a very practical move toward high-resolution graphics. The bad news is this: With a 30% larger bit map to manipulate, software running on the EEGA will be slower than EGA software. Among other implications, this makes it imperative that EGA software developers write the highest-performance code possible.

Not all the graphics at Comdex were EGA-related, though, by a long shot. There were a number of IBM Professional Graphics Controller (PGC) compatible systems (in fact, Zenith was demonstrating its 80386-based system with a Turbo PGC, which was interesting since virtually none of the speed of the graphics was attributable to the 386). There were systems built around both the Intel 82786 and the TI 34010 (which I'm planning to look at soon), and there were proprietary systems with resolutions up to 1K-by-1K. Unfortunately, there were no signs of standardization for graphics above the EEGA level. I suppose standardization will have to wait for either the TI or the Intel chip to become a mass-market success. (Or some other chip—it will happen eventually). Another possibility is that either or both chips may prove to be powerful enough to support good performance through device drivers, in which case we may (finally!) be near the era of true device independence.

IBM could, of course, set the next graphics standard with its rumored Turbo EGA, or with another graphics adapter. However, it doesn't seem to be as easy as it used to be for IBM to set standards; the PGC was hardly an unqualified success, and the EGA became a standard only because third party manufacturers imitated the technology and brought the price way down. It's not at all clear that if IBM were to come out with a new adapter tomorrow it would have much impact, particularly if the technology were sufficiently proprietary that other manufacturers couldn't clone it. For the near future, at least, fragmentation seems to be the nature of the high end of the PC graphics market.

On the bright side, many of the high-end display adapters featured EGA compatibility. Whatever direction the graphics market takes in the next few years, it's safe to say that EGA software will retain its value, particularly if that software can support 640 x 480 graphics.

EGA Programming: ALUs and Latches

I'm going to begin our detailed tour of the EGA at the heart of the flow of data through the EGA: the four ALUs built into the EGA's Graphics Data Controller (GDC) chips. The ALUs (one for each display memory plane) are capable of ORing, ANDing, and XORing CPU data and display memory data together, as well as masking off some or all of the bits in the data from affecting the final result. All the ALUs perform the same logical operation at any given time, but each ALU operates on a different display memory byte. (Recall that the EGA has four display memory planes, with one byte in each plane at any given display memory address. All four display memory bytes operated on are read from the same address, but each ALU operates on a byte that was read from a different plane and writes the result to that plane.) This arrangement allows four display memory bytes to be modified by a single CPU write (which must normally be preceded by a single CPU read, as we will see). The benefits are clear: If the CPU had to select each of the four planes in turn via OUTs and perform the four logical operations itself, EGA performance would be slowed to a crawl.

Figure 1 is a simplified depiction of data flow around the ALUs. Each ALU has a matching latch that holds the byte read from the corresponding plane during the last CPU read from display memory—even if that particular plane wasn't the plane the CPU actually read on the last read access. (Only one byte can be read by the CPU with a single display memory read; the plane supplying the byte is selected by the Read Map register. However, the same address from all four planes is always read when the CPU reads display memory, and those four bytes are held in their respective latches.)

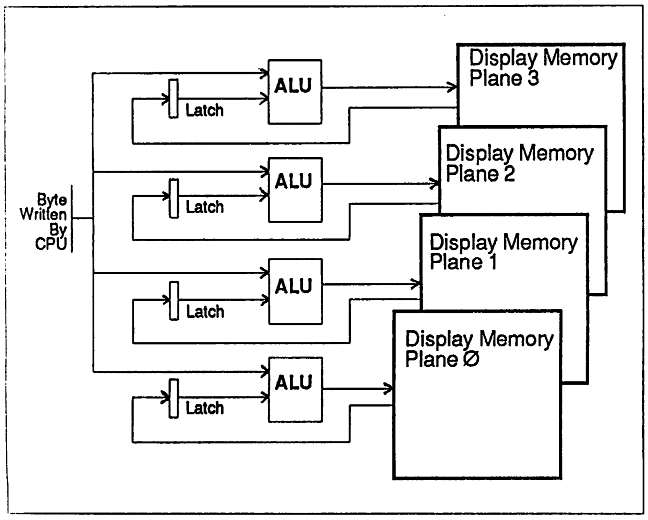


Figure 1: Simplified Data Flow Around the Arithmetic Logic Units (ALUs) in the Graphics Data Controller

Each ALU logically combines the byte written by the CPU and the byte stored in the matching latch, according to the setting of bits 3 and 4 of the Data Rotate register (and the Bit Mask register as well, which I'll cover next time), and then writes the result to display memory. It is most important to understand that neither ALU operand comes directly from display memory. The temptation is to think of the ALUs as combining CPU data and the contents of the display memory address being written to, but they actually combine CPU data and the contents of the last display memory location read, which need not be the same location being modified. The most common application of the ALUs is indeed to modify a given display memory location, but doing so requires a read from that location to load the latches before the write that modifies it. Omission of the read results in a write operation that logically combines CPU data with whatever data happens to be in the latch from the last read, which is normally undesirable.

Occasionally, however, the independence of the latches from the display memory location being written to can be used to great advantage. The latches can be used to perform 4-byte-at-a-time (one byte from each plane) block copying; in this application, the latches are loaded with a read from the source area and written unmodified to the destination area. The latches can be written unmodified in one of two ways: by selecting write mode 1 (for an example of this, see the "On Graphics" article in the last issue of *PJ*) or via the Bit Mask register.

The latches can also be used to draw a fairly complex area fill pattern, with a different bit pattern used to fill each plane. The mechanism for this is: Generate the desired pattern across all planes at any display memory address. Generating the pattern requires a separate write operation for each plane so that each plane's byte will be unique. Read that memory address to store the pattern in the latches. The contents of the latches can now be written to memory any number of times by using either write mode 1 or the bit mask since they will not change until a read is performed. If the fill pattern does not require a different bit pattern for each plane, filling can be performed more easily by using the Map Mask register (and, if display memory isn't zeroed, the Set/Reset registers, of which, more next time).

The sample program in Listing 1 fills the screen with horizontal bars, then illustrates the operation of each of the four ALU logical functions by writing a vertical 80-pixel-wide box filled with solid, empty, and vertical and horizontal bar patterns over that background using each of the functions in turn. When observing the output of the sample program, remember that all four vertical boxes are being drawn with exactly the same code—only the logical function that is in effect differs from box to box.

You will observe that all graphics in the sample program are done in black-and-white by writing to all planes since this shows the operation of the ALUs most clearly. Selective enabling of planes would produce color effects; in this case, the operation of the logical functions must be evaluated on a plane-by-plane basis because only the planes enabled by the Map Mask register will be affected.

Logical function 0, data unmodified, is the standard mode of operation of the ALUs. In this mode, the CPU data is combined with the latched data by ignoring the latched data entirely. Expressed as a logical function, this could be considered CPU data ANDed with 1 (or ORed with 0). This is the mode to use whenever you want to place CPU data into display memory, replacing the previous contents entirely. It may occur to you that there is no need to latch display memory at all when the data unmodified function is selected. In the sample program, that is true, but if the bit mask is being used, the latches must be loaded even for the data unmodified function, as I'll discuss in the next article.

Logical functions 1 through 3 cause the CPU data to be ANDed, ORed, and XORed with latch data, respectively. Of these, XOR is the most useful because exclusive-ORing is a traditional way to perform animation. The uses of the AND and OR logical functions are less obvious. AND can be used to mask a blank area into display memory or to mask off those portions of a form that don't overlap an existing display memory image. OR could conceivably be used to force an image into display memory over an existing image. To be honest, though, I have yet to come up with a truly valuable application for the OR logical function; if you've got one, write and let me know what it is.

Notes on the ALU/LATCH Sample Program

EGA settings such as the logical function select should be restored to their default conditions before the BIOS is called to output text or draw pixels. The EGA BIOS does not guarantee that it will set most EGA registers except on mode sets, and there are so many compatible BIOSes around that the code of the IBM BIOS is not a reliable guide. For instance, when the BIOS is called to draw text, it's likely that the result will be illegible if the Bit Mask register is not in its default state. Similarly, a mode set should generally be performed before exiting a program that tinkers with EGA settings.

Along the same lines, the sample program does not explicitly set the Map Mask register to ensure that all planes are enabled for writing. The mode set for mode 10 hex leaves all planes enabled, so I did not bother to program the Map Mask register, or any other register besides the Data Rotate register, for that matter. However, the profusion of compatible BIOSes means there is some small risk in relying on the BIOS to leave registers set properly. There are only a few clone chip sets, and they all seem to be compatible, so for the highly safety-conscious, the best course would seem to be to program data control registers such as the Map Mask and Read Mask explicitly before relying on their contents.

(On the other hand, any function the BIOS provides explicitly—as part of the interface specification—such as setting the palette RAM, should be used in preference to programming the hardware directly.)

The code that draws each vertical box in the sample program reads from display memory immediately before writing to display memory. The read operation loads the EGA latches. The value read is irrelevant as far as the sample program is concerned; it is necessary to perform a read to load the latches, and there is no way to read without placing a value in a register. This is a bit of a nuisance since it means that the value of some 8-bit register must be destroyed. Under certain circumstances, a single logical instruction such as XOR can be used to perform both the read to load the latches and the write to modify display memory without affecting any register, but since this requires the bit mask, it'll have to wait until next time.

All text in the sample program is drawn by EGA BIOS function 13 hex, write string. This function is also present in the AT's BIOS, but not in the XT's or PC's, and as a result is rarely used; the function is always available if an EGA is installed, however. Text drawn with this function is relatively slow. If speed is important, a program can draw text directly into display memory much faster in any given display mode. The great virtue of the BIOS write string function in the case of the EGA is that it provides an uncomplicated way to get text on the screen reliably in any mode and color, over any background.

The expression used to load DX in the TEXT_UP macro in the sample program may seem strange, but it's a convenient way to save a byte of program code and a few cycles of execution time. DX is being loaded with a word value that's composed of two independent immediate byte values. The obvious way to implement this would be with

```
MOV DL,VALUE1
MOV DH,VALUE2
```

which requires four bytes and eight cycles on the 8088. If you shift the value destined for the high byte into the high byte with MASM's shift-left operator SHL (*100h would also work) and then logically combine the values with MASM's OR operator (or the ADD operator), both halves of DX can be loaded with a single instruction, as in

```
MOV DX,(VALUE2 SHL 8) OR VALUE1
```

which takes only three bytes and four cycles. As shown, a macro is an ideal place to use this technique; the macro invocation can refer to two separate byte values, making matters easier for the programmer, while the macro itself can combine the values into a single word-sized constant.

A minor speed tip illustrated in the listing is the use of INC AX and DEC AX in the DrawVerticalBox subroutine when only AL actually needs to be modified. Word-sized register increment and decrement instructions are only one byte long and execute in 2 cycles on an 8088, while byte-sized register increment and decrement instructions are two bytes long and execute in 3 cycles. Consequently, when speed counts it is worth using a whole 16-bit register instead of the low 8 bits of that register for INC and DEC—if you don't need the upper 8 bits of the register for any other purpose.

Another speed tip is the use of registers to temporarily store all constants in the background-drawing loop. Moving a constant to a 16-bit register takes a 3-byte instruction and adding a constant to a 16-bit register takes a 4-byte instruction, while moving and adding 16-bit registers to 16-bit registers require only 2-byte instructions. The register-register operations are also at least 1 cycle faster; they are probably a good deal faster than that relative to the constant operations because of the prefetch queue, a topic I will return to later.

The latches and ALUs are central to high-performance EGA code because they allow programs to process across all four memory planes without a series of OUTs and read/write operations. It is not always easy to arrange a program to exploit this power, however, since the ALUs are far more limited than a CPU. In many instances, however, additional hardware in the EGA, including the bit mask, the set/reset features, and the barrel shifter, can assist the ALUs in controlling data. I will begin to examine these features next time.

Puzzles

I've got a couple of unsolved (in one case, partially solved) puzzles this month.

First, another MASM problem. When I use macros with several arguments, the text often runs past column 80. The code still assembles properly, but printouts are a mess. Does anyone know how to continue a macro invocation (or a STRUC initialization, for that matter) on the next line?

The second puzzle pops up regularly in coding high-performance graphics drivers—how the heck do you determine just what's the fastest 8088 code for a given purpose? Hand-optimization of Assembler code is always an art, but the prefetch queue makes 8088 optimization black magic. (See my article "More Optimizing For Speed," *PJ*, July/August 1986, for an introduction to prefetch queue effects.) Optimizing 80286 code is somewhat less critical, both because the 8088 is far more common right now and because ATs are faster than PCs no matter how fast 8088-oriented code is, but 80286 optimization is becoming more important, so that's an issue too. And, of course, 80386 optimization looms on the horizon.

I ran across a particular optimization puzzle while implementing Bresenham's line-drawing algorithm: how best to rotate a pixel mask (one bit set and the others reset) to the right (moving one pixel to the right), incrementing the screen address by one when the set bit in the pixel mask wraps around from bit 0 to bit 7? The standard code for this instance would be

```
        ROR AL,1
        JNC NO_INC
        INC DI
NO_INC:
```

where AL is the pixel mask (with only one bit set to 1), and DI is the screen address of the byte containing the next pixel to be drawn. However, JNC is executed 7 of 8 times the above code is executed, requiring 16 cycles each time, for an average code execution time for the conditional increment portion of this code of ((16 * 7) + (4 + 2)) / 8 = 14.75 cycles on the 8088. (The conditional increment code is JNC NO_INC / INC DI; the ROR AL,1 instruction is not counted because I'm interested only in the relative conditional increment timings of the approaches I'll examine.) On the other hand, the less elegant looking code:

```
        ROR AL,1
        JC DO_INC
CONTINUE_AFTER_INC:
        :
        :
DO_INC:
        INC DI
        JMP CONTINUE_AFTER_INC
```

where DO_INC is outside the loop the rotate is performed in but within range of a short jump, requires only ((7 * 4) + (16 + 2 + 15)) / 8 = 7.6 cycles to perform the conditional increment on average. (Thanks to Dan Illowsky for this unorthodox but effective approach.) The speed of this code is a result of the vast difference on the 8088 between a conditional jump taken and one not taken.

The fastest code, however, is that which eliminates branching entirely. To wit:

```
        ROR AL,1
        ADC DI,0
```

Continued...

TURBO SOURCE SEARCH (TSS)

TSS is a **RBBS-by-mail** for **TURBO** Pascal. Stop paying AT&T, MCI, SPRINT et al. 10 times what it will cost you if you get it by mail from TSS.

Join **TSS** for $25. Then buy our library disks, **$2 per disk.** (45 disks and growing!!) Archived files = **700+ Kb** of code per **TSS** disk. **TSS** gives more bytes/buck! **Why pay more to get less?**

For your $25 you get
- 2 **FREE** library disks (utilities & catalog)
- 2 **FREE TSS** disks with your first order
- a validated password for the **online TSS-BBS**

TSS-BBS will be an "e-mail" drop-off for members to place orders then log off. It will also serve more traditional RBBS needs. Members will be able to DL the **TSS** disks **(at 10 times the cost of mails!)** if they really MUST have it now!

Join the **TSS BBS-thru-the-mail** for the best in public domain and shareware **TURBO** code.

TURBO SOURCE SEARCH
P.O. Box 876, Scituate, MA 02066, (617) 545-6677
payment: VISA/MC/COD if validated, otherwise check/money orders

Number 35 on the Reader Service Card

$99 BUSINESS BOARD

24 hour business information center using modems at 3ØØ/12ØØ/24ØØ baud. Has a multi-level menu system that is easily customized with no programming req'd. Includes remote PC operations and integrated data base management. Source code additional charge.

$99 PC COMMX $119 CP/M

Emulates VT1ØØ,Wyse,HP,ADM,TV,IBM,ADDS,file transfers: KERMIT, XMODEM, COMMX mainframe, Telex/TWX. Instant DOS/foreground switch! Unattended macro controls and 7ØØ entry dial directory. Electronic mail sub-system!

$59 C DATA ENCRYPTION

Data Encryption Standard (U.S. government standard FIPS PUB46) in Microsoft "C". Includes compression & telecomm formatting, allowing for faster transmission & storage on any computer or service. Complete "C" source code provided for additional $249.

HAWKEYE GRAFIX Inc
Box 14ØØ, Oldsmar Florida 33557
Call 813-786-8161

Number 57 on the Reader Service Card

requires exactly 4 cycles to perform each conditional increment—more than 3 times faster than the original code. This is an excellent example of the way in which the quirks of the 8088 make it possible to hand-optimize small portions of Assembler code to a far greater extent than is true of most processors.

So what's the puzzle? The puzzle is that I still don't know what's really the fastest way to perform the above rotate and conditional increment. The performance estimates above were arrived at by adding the execution times Intel publishes, but the published times assume that the prefetch queue isn't empty. For example, since ROR AL, 1 / ADC DI,0 constitutes 6 bytes, since the 8088 prefetch queue is only 4 bytes long, and since 4 cycles are required to fetch each instruction byte, there's no way that particular code fragment could ever execute in only the specified 6 cycles, even if the prefetch queue were full when ROR began executing. If the prefetch queue is empty, the ADC DI,0 instruction alone will require 16 cycles (4 cycles times 4 bytes), a far cry from the 4 cycles assumed above, and, in fact, exactly as long as the nominal time required to make a conditional jump. The time required to jump is unclear as well because the prefetch queue is emptied by each jump and may take additional time beyond the 16 cycles specified to refill before the instruction branched to can start execution. All the code fragments above are likely to take somewhat longer than specified; just how much longer is the question.

So. Here's what I'd like to know (and share with *PJ*'s readers). First, what is the precise overhead of branching and flushing the queue? Are additional cycles beyond the published timings required to get enough bytes to begin executing the next instruction?

Second, how long does a conditional jump not taken take to execute; is the second byte of the instruction even read? If so, is it read while the jump condition is evaluated?

Third, how important, in general, is the prefetch queue in determining performance of 80286/80386 code? The 80286 can fetch twice as many instruction bytes per memory access as the 8088, but it also executes most instructions in many fewer clock cycles and so uses up instruction bytes at a faster rate.

Fourth (and most important), what techniques have *PJ* readers come up with for evaluating the performance of time-critical code? Ideally, Intel would make available all the information about the 8088's microcode required to accurately determine execution times, but I know of no such publication. There are a number of code profilers out there, but most of them measure gross effects by reporting overall time spent in a given routine. To truly optimize, it's necessary to understand where every cycle goes, and to do that, each instruction in the innermost time-critical loops has to be examined in the context of normal operation. There's hardware out there that can do this; sadly, such equipment is outside the price range of many developers, especially individuals. On behalf of the under-funded developers of the world, I'd love to hear about a way to do cycle-by-cycle in-context performance analysis in software.

Who cares about a few cycles here and there? In a time-critical code path, any programmer who cares about turning out a superior product should, and on the PC, virtually all real-time graphics are time-critical. Let us know how you do it. Make the world of 8088 programming a little better.

Next Time

I'm going to explore a couple of interesting EGA topics in the next article. I'll continue examining the data flow around the ALUs, including the bit mask, data rotate, and set/reset capabilities, which are crucial to pixel-aligned graphics and fast color control. I'll also talk about color versus monochrome operation and downward compatibility with the CGA, MDA, and Hercules Graphics Card, plus the usual discussion of interesting hardware and/or software I've come across. See you then. □

Michael Abrash is a Senior Software Engineer for Orion Industries of Redwood City, CA, a manufacturer of PC-based instrumentation and microprocessor development systems.

Code follows.

Here's why you should choose Periscope as your debugger...

You'll get your programs running fast. "It works great! A problem we had for three weeks was solved in three hours," writes Wade Clark of MPPi, Ltd.

You'll make your programs solid. David Nanian says, "I can't live without it!! BRIEF, a text editor my company wrote, would not be as stable as it is today without Periscope."

You'll protect your investment. We won't forget you after the sale. You'll get regular software updates, including a FREE first update and notice of later updates. You'll get technical help from Periscope's author. And you'll be able to upgrade to more powerful models of Periscope if you need to. One Periscope user writes, "... your support has won over even the heart of this hardened programmer!"

You deserve the best. Thousands of programmers rely on the only debugger that PC Tech Journal has ever selected as **Product of the Month** (1/86). You owe it to yourself to find out why, first hand.

You can try it at no risk. You get an unconditional 30-Day, Money-Back Guarantee, so you can't lose.

Start saving time and money now — order toll-free, 800/722-7006. Use MasterCard, Visa, COD, or a qualified company purchase order. As one user puts it, Periscope is "one of the rare products, worth every penny!"

New Version 3 is better than ever!

Periscope I, software, manual, protected memory board and breakout switch	$345
Periscope II, software, manual, and breakout switch	$175
Periscope II-X, software and manual	$145

Add shipping - $3 US; $8 Canada; $24 elsewhere.

The PERISCOPE Company, Inc.

14 Bonnie Lane • Atlanta, GA 30328 • 404/256-3860

Number 6 on the Reader Service Card

EGA Data Control

by
Michael Abrash

This month I'm going to cover a lot of ground. First, I'll discuss the monitors the EGA can support and look into issues of EGA compatibility with other adapters. Next, I'll continue our detailed exploration of the data management capabilities of the GDC chip. Finally, I'll look at three EEGAs.

Color, Monochrome, and Compatibility

I've become aware that few people understand exactly what displays EGAs can drive and what adapters EGAs are compatible with. I'm going to do my best to remedy that here.

A standard EGA can drive a monochrome display, a color display, or an enhanced color display. Monochrome operation is simpler, so I'll start with that.

When driving a monochrome display, the EGA supports a text mode identical in appearance to that of the Monochrome Display Adapter (MDA), with characters in 9 x 14 boxes. The bit map is at B000:0000, just as with the MDA, and attributes can be (and are set up by the BIOS to be) interpreted in the same way as with the MDA. The EGA is BIOS-compatible with the MDA, but the registers of the EGA are only partially compatible. Most importantly, the cursor location register is MDA-compatible. The cursor start and stop scan line registers are not fully compatible, but they can be set reliably through the BIOS, which is compatible across the MDA, CGA, and EGA. Given the way in which the MDA is normally programmed, the EGA can be considered to be highly MDA-compatible. In particular, if registers other than the cursor location aren't set directly, the EGA is a fine MDA replacement. Since the MDA supports only one mode, mode 7, there is little reason to directly program MDA registers other than the cursor location, so the EGA can generally be substituted for the MDA.

When configured for monochrome operation, the EGA provides a graphics mode with a resolution of 640 x 350, with attributes of black, white, high-intensity, and blink. This mode is similar to the 640 x 350 color graphics mode except that two planes, rather than four, are used. Unfortunately, the EGA's monochrome graphics mode is not even remotely compatible with the graphics mode of the Hercules Graphics Card (HGC). It is possible to program an EGA to support the same bit map as the HGC does (someday I'll show how in this space). Since, however, the HGC has no BIOS support for mode switches, all software that supports the HGC programs the hardware directly; because the EGA is not fully register-compatible with the HGC, this will not work with the EGA. So, practically speaking, the standard EGA is compatible with the MDA but not the HGC.

Color EGA operation is more complex. The EGA can be configured to support only 200-scan-line display modes, or it can be configured to support both 200- and 350-scan-line modes. (In the configuration switch table for the EGA there appear to be more choices, but the only important choice is between 200 and 350 scan lines. There is no reason to power up an EGA in 40-column mode since it has no composite or RF output and the high-resolution emulation mode is identical to the 200-scan-line 80-column setting.) In 200-scan-line operation, the EGA has two new graphics modes (unsupported by the CGA), 320 x 200 in 16 colors and 640 x 200 in 16 colors. In 350-scan-line operation, the EGA has a 640 x 350 16-color graphics mode; additionally, in 350-scan-line operation text in 80 x 25 text mode is in an 8 x 14 box, rather than in the coarse 8 x 8 box of the CGA. High-resolution text is far superior to CGA text, but there is one important limitation—border colors other than black aren't supported in 350-scan-line modes (due to tight retrace timings). With that background out of the way, let's look at EGA emulation of the CGA.

In general, the EGA is only a fair Color/Graphics Adapter (CGA) replacement. The EGA supports all the display modes of the CGA, as well as all CGA bit maps at B800:0000. The EGA is also BIOS-compatible with the CGA, but, unlike the MDA's, the CGA's hardware is often programmed directly, and the EGA is not fully register-compatible. Most notably, the Color Select register, which is used to select among the CGA's palettes and background colors, is not present in the EGA. A BIOS function is provided to set these registers, but it supports only one of the two palette intensities provided by the CGA. Also, border colors in high-resolution modes aren't supported. In order to write a program that would run on both the CGA and EGA, you must avoid all accesses to registers other than the cursor location and start address registers, again including the cursor start and stop registers. This is possible, but the differences in color capabilities and color setting techniques between the two adapters mean that a program that would run on both adapters would be limited to two three-color palettes in graphics modes, with a black border only. In short, the EGA can be used to run BIOS-compatible CGA software, but the two boards are really quite different.

A summary of the standard EGA's compatibility with existing display adapters: CGA and MDA programs that do not access registers other than the cursor location and start address registers directly should run on the EGA. Bit-map compatibility is maintained by the EGA. HGC programs will not run on a standard EGA although it is possible to program an EGA to support an HGC-format bit map.

Many EGAs have extended CGA and/or HGC compatibility built in. These boards can create headaches for developers, for many of them are not really hardware-compatible with the

CGA or HGC. A common method of implementing compatibility is to generate an NMI (non-maskable interrupt) on attempted access to the CGA's (or HGC's) registers, with the NMI-handling routine forcing the CRT registers to settings that match the setting of the CGA mode register. This approach means that software that reprograms the 6845 CRT of the CGA (to get extra rows and columns, for example, or to smooth scroll) will not work on EGAs with NMI-based compatibility.

There are, incidentally, other problems with boards that provide downward compatibility—in particular that it can be difficult to determine what type of adapter a program should be trying to drive; for example, a program that looks for the string "IBM" in the EGA BIOS to see whether an EGA is installed (a common, although crude, method) will attempt to drive an EGA even if emulation is active and the program should be trying to drive an emulated CGA. One approach to this problem is letting the user select the display type at installation time.

As a result, the recommendation that developers leave registers other than the cursor location and start address registers alone and go through the BIOS holds even on CGA- and HGC-compatible EGAs. Bit maps are the same, so it's all right to go directly to display memory (fortunately, since the BIOS write and read dot functions are so slow as to be virtually useless). It's too bad that the BIOS doesn't provide full support for CGA palettes, though, and that border colors aren't supported in 350-scan-line modes.

Of course, if you're not interested in having your applications run on anything but an EGA (or anything but a CGA), or if you're willing to write your programs to specifically support a number of display adapters, you can go right to the hardware. I don't mean to say that you shouldn't use fast, nonstandard techniques when you have to—just be aware that as the world of display adapters expands, with CGAs, MDAs, HGCs, EGAs, EEGAs, and new display technologies that provide downward compatibility to varying degrees, it's going to be harder to support them all directly. So. . .where you can afford to, go through the BIOS; go directly to the hardware when you can't live without it.

There's one other compatibility issue concerning the EGA, and that's what displays the EGA can coexist with. The answer is that an EGA configured for color can be in the same system with an MDA (or HGC with up to 32K available), and an EGA configured for monochrome can be in the same system with a CGA. This is possible because a color EGA uses registers in the range 3C0–3DC and a monochrome EGA uses registers in the range 3B0–3CF, complementing the address ranges of the MDA and CGA, respectively. Two EGAs, one color and one monochrome, cannot coexist because both would use registers in the 3CX range. It is possible to jumper an EGA to set its address range to 2XX rather than 3XX, allowing two EGAs to be installed together without conflict; however, no current BIOS supports this alternate address range, so a good deal of work would be required in order to use two EGAs together in this fashion.

A Closer Look at the GDC

Last time, we looked at a simplified model of data flow within the GDC chips of the EGA, featuring the latches and ALUs. This month I'll expand that model to include the barrel shifter and bit mask, leaving only the set/reset capabilities to be explored.

EGA Data Rotation

Figure 1 shows the expanded model of GDC data flow. First, I'd like to look at the barrel shifter. (A barrel shifter is circuitry capable of shifting—or in the EGA's case, rotating—data an arbitrary number of bits in a single cycle.) The barrel shifter can rotate incoming CPU data up to seven bits to the right (toward the least significant bit), with bit 0 wrapping back to bit 7. Thanks to the nature of barrel shifters, this rotation requires no extra processing time over normal, unrotated EGA operation. The number of bits by which CPU data is shifted is controlled by bits 2–0 of GDC register 3, the Data Rotate register, which also contains the ALU function select bits.

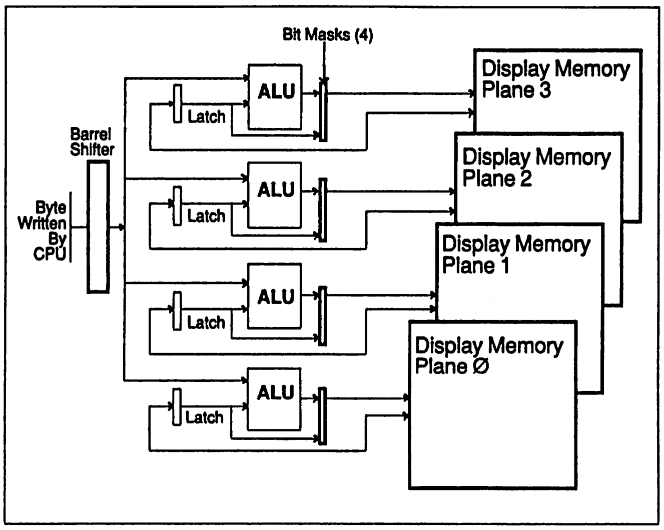


Figure 1: Data flow through the GDC (Set/Reset circuitry not shown)

The barrel shifter is powerful, but it sounds more useful than it really is. This is because the GDCs can rotate only CPU data, a task which the CPU itself is perfectly capable of performing. Two OUTs (one to set the GDC index register and one to set the Data Rotate register) are needed to select a given rotation, and often it's easier and/or faster to simply have the CPU rotate the data of interest CL times than it is to set the Data Rotate register. If only the EGA could rotate latched data, there would be all sorts of useful applications for rotation, but, sadly, only CPU data can be rotated.

The drawing of bit-mapped text is one good use of the barrel shifter, and I'll demonstrate that application below. In general, though, don't knock yourself out trying to figure out how to work data rotation into your programs—it just isn't all that useful.

The Bit Mask

The EGA has a bit mask for each of the four memory planes. Figure 2 illustrates the operation of one bit of a bit mask. This circuitry occurs eight times in the bit mask for a given plane, once for each bit of the byte written to display memory. Briefly, the bit mask determines on a bit-by-bit basis whether the source for each byte written to display memory is the ALU for that plane or the latch for that plane. The bit mask is controlled by GDC register 8, the Bit Mask register. If a given bit of the Bit Mask register is 1, the corresponding bit of data from the ALUs is written to display memory, while if that bit is 0, the corresponding bit of data from the latches is written to display memory unchanged.

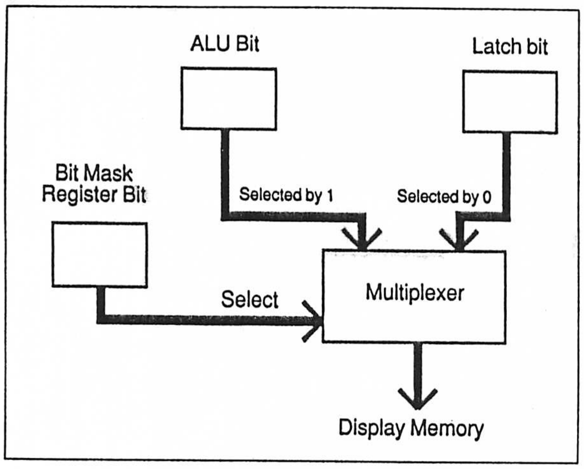


Figure 2: Operation of one bit of the Bit Mask register

The most common use of the bit mask is to allow updating of selected bits within a display memory byte. This works as follows: The byte of interest is latched; the bit mask is set to preserve all but the bit or bits to be changed; the CPU writes to display memory with the bit mask preserving the indicated latch bits and allowing ALU data through to change the other bits. Remember, though, that it is not possible to alter selected bits in a display memory byte directly; the byte must first be latched by a CPU read, and then the bit mask can keep selected bits of the latched byte unchanged.

Listing 1 shows a program that uses the bit mask and data rotation capabilities of the GDC to draw bit-mapped text at any screen location. (The BIOS draws only characters on character boundaries; in 640 x 350 graphics mode the default font is drawn on byte boundaries horizontally and every 14 scan lines vertically.) Note that the 8 x 8 BIOS font is used; a pointer to this font is obtained through function 11h of INT 10h.

The bit mask can be used for much more than bit-aligned fonts. For example, the bit mask is useful for fast pixel drawing, such as that performed when drawing lines. At some point in the future, I'll demonstrate such applications. The example in Listing 1 is designed to illustrate the use of the Data Rotate and Bit Mask registers and is not as fast or as complete as it might be. The case where text is byte-aligned could be detected and performed much faster, without the use of the Bit Mask or Data Rotate registers and with only one display memory access per font byte, rather than four. Also, it would be faster to use a table lookup to calculate the bit masks for the two halves of each character than the shifts used in the example.

For another (and more complex) example of drawing bit-mapped text on the EGA, see John Cockerham's article, "Pixel Alignment of EGA Fonts," *PC Tech Journal*, January 1987. Parenthetically, I'd like to pass along John's comment about the EGA: "When programming the EGA, everything is complex." He's got a point.

Of Interest

I've gotten the opportunity to try out several EEGAs, with impressive results. If you've got a monitor capable of going up to 640 x 480 or higher, you should consider an EEGA, and if you're developing an application that needs higher resolution (what application doesn't?), you should certainly take a long look at supporting one or more EEGAs. As we'll see, all EEGAs are not alike; each of the three I'll look at has unique features.

Before I begin, I'd like to make it clear that the following is not a review in any conventional sense. I'm not trying to cover the boards in every detail because I'm not trying to help you decide which to buy. By the same token, I'm not trying to blanket the field—you won't find comparison charts of 32 EGAs here. What I am trying to do is help you decide whether you should support the special features of these boards, or whether one of them might help you solve special hardware needs. You might think of topics discussed under the "Of Interest" heading as overviews of advanced display technology that I think developers should know about.

The VEGA Deluxe from Video-7 (550 Sycamore Drive, Milpitas, CA 94035, (408) 943-0101) is the follow-up to the extremely popular VEGA, one of the boards that fueled the first great boom in the EGA market. The VEGA Deluxe is built around the Chips & Technology EGA chip set, which has proved to be highly IBM EGA-compatible. The VEGA Deluxe supports resolutions of 640 x 480 and 752 x 410 although the only means of accessing these resolutions in the base package is via Windows drivers. (Late note: Video-7 indicates that other drivers are on the way and that developer's information is available.) The VEGA Deluxe also features full CGA and HGC compatibility. Given Video-7's stature in the EGA market, it would make sense for any developer who is going to support EEGAs to include the VEGA Deluxe.

The EGA 2001/PLUS, from Ahead Systems (1977 O'Toole Ave., Suite B105, San Jose, CA 94131, (408) 435-0707) is also built around the C & T chip set. The 2001/PLUS offers full CGA and HGC compatibility and adds a nice touch—HGC compatibility on color monitors. A 640 x 480 mode is supported (I propose this as a minimum requirement for a board to be considered an EEGA), as well as 720 x 396, which provides IBM S3G graphics compatibility. 80 x 66/45/43/34/25 and 132 x 44/28/25 text modes are provided; these modes provide IBM 3278/79 MOD 2–5 compatibility, as well as compatibility with many popular PC extended spreadsheet and text editing modes. The board is a compact, clean half-card. In short, the

2001/PLUS is a well-conceived and executed board that covers a lot of ground, and it's worth looking at if your requirements exceed the standard EGA.

The EVA 480, from Tseng Laboratories (205 Pheasant Run, Newtown, PA 18940, (215) 968-0502) sports a number of extras. Like the other EEGAs, the EVA 480 offers 640 x 480 graphics and complete downward compatibility with the CGA and HGC, as well as 132 x 44/28/25 and 80 x 60/43/25 text and Windows, Autocad, and Halo drivers. Because it is built around Tseng Labs' proprietary ET2000-series chip set (which, incidentally, I participated in the design of), the EVA 480 sports two unique extras: a zoomable hardware window and faster memory access.

The EVA 480 features a built-in hardware window (in addition to the standard EGA split screen), which can start and stop on any scan line vertically and on any byte boundary horizontally. This window can display the contents of any area of EGA memory, allowing instant popups, for example. The window area can be zoomed by pixel replication in both directions in graphics modes and horizontally in text mode, by a factor of up to 8 times horizontally and 16 times vertically. This could be very handy for a developer who needs to blow up part or all of the screen quickly.

The EVA 480 also allows the CPU much greater access to display memory than does the standard EGA. The standard EGA allows one CPU access to display memory every four character clocks while the EVA 480 allows one access every character clock. During highly display-memory-intensive operations, the EVA 480 is noticeably quicker than other EGAs. For example, I ran ATPERF, *PC Tech Journal*'s 286 Performance and Compatibility Suite ("Out from the Shadow of IBM . . ." by Steven Armbrust, Ted Forgeron, and Paul Pierce, *PC Tech Journal*, August 1986), on a 10-MHz AT clone in order to determine how many wait states were being inserted by each of the three EEGAs I've discussed. Both the VEGA Deluxe and the EGA 2001/PLUS performed at the same speed, inserting 18 wait states on average—not surprising given that they're built around the same chip set. The EVA 480 inserted only 9 wait states, the minimum number of wait states an 8-bit device can cause during a 16-bit access in an AT. For programs that access display memory less frequently, the gain will be less, but the EVA 480 is clearly the faster EGA.

Tseng Labs is the first but not the only company to make a higher performance EGA chip set. Chips and Technologies is coming out with a chip set with all sorts of extensions to the EGA, including more frequent CPU access to display memory. I'll discuss this chip set in more detail next time.

Don't expect the relatively sluggish performance of the EGA to improve much beyond the EVA 480, though, no matter what chip sets appear. Improving CPU memory access is about all that can be done without losing EGA compatibility, and right now compatibility is essential. Moreover, the higher performance from improved CPU memory access barely balances the greater size of the bit maps in the EEGA's highest-resolution modes.

On Performance

I'm running short on space, so all I'm going to say about performance is to suggest that you read "High-Performance Software Analysis on the IBM PC" by Byron Sheppard (*BYTE*, January 1987). It looks to me like Sheppard's routines provide very good software-based performance analysis. To the extent that I've tried them out, they've worked well and given pretty much the results I'd expect (although I will say that figur-

Introducing

Quaid Analyzer

the tool that created CopyWrite

Now you can debug your own programs with a professional quality debugger - the one that unraveled every form of copy-protection used on the PC.

With the Quaid Analyzer, you can:

- ☐ See occurences of any interupt, with its meaning shown on the screen.
- ☐ View memory as text or instructions, scrolling as easily as you do with an editor.
- ☐ Run until a memory location or I/O port is changed.
- ☐ Protect your hard disk from accidental destruction.
- ☐ Analyze software without the source, even when it uses countermeasures to thwart tracing.
- ☐ See all stages of the boot load.

We kept the Quaid Analyzer off the market to avoid helping publishers with copy-protection. Now that copy-protection is gone, we can sell it to you.

The Quaid Analyzer is a software tool occupying 100K bytes. It runs on any IBM PC and most MS-DOS systems without hardware modification.

To order Call **(416) 961-8243**

Quaid Analyzer $99 U.S.

All orders shipped at our expense within a day. All major credit cards accepted.

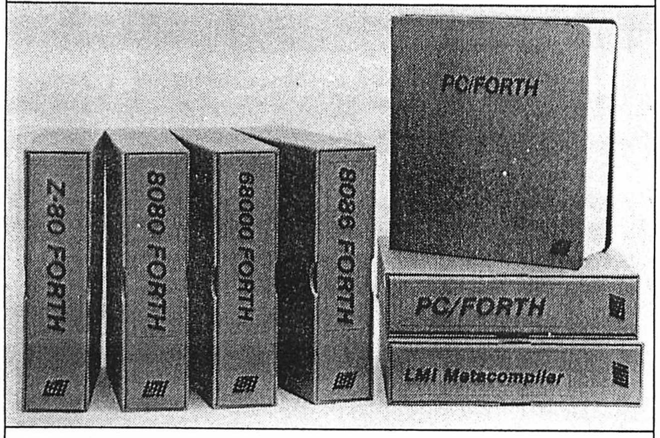
or return coupon to:
45 Charles St. East
Third Floor, Dept. 603
Toronto, Ontario. M4Y 1S2

Payment method MC-Visa-Amex-Diners-Check
Card No. ____
Expiry Date ____
Name ____
Address ____
City/State ____
Phone No. ____
Signature ____

Quaid Software Limited

Ask about Disk Explorer the program that takes over where Quaid Analyzer leaves off.

TOTAL CONTROL with *LMI FORTH*™



For Programming Professionals:

an expanding family of compatible, high-performance, Forth-83 Standard compilers for microcomputers

For Development:
Interactive Forth-83 Interpreter/Compilers
- 16-bit and 32-bit implementations
- Full screen editor and assembler
- Uses standard operating system files
- 400 page manual written in plain English
- Options include software floating point, arithmetic coprocessor support, symbolic debugger, native code compilers, and graphics support

For Applications: Forth-83 Metacompiler
- Unique table-driven multi-pass Forth compiler
- Compiles compact ROMable or disk-based applications
- Excellent error handling
- Produces headerless code, compiles from intermediate states, and performs conditional compilation
- Cross-compiles to 8080, Z-80, 8086, 68000, 6502, 8051, 8096, 1802, and 6303
- No license fee or royalty for compiled applications

For Speed: CForth Application Compiler
- Translates "high-level" Forth into in-line, optimized machine code
- Can generate ROMable code

Support Services for registered users:
- Technical Assistance Hotline
- Periodic newsletters and low-cost updates
- Bulletin Board System

Call or write for detailed product information and prices. Consulting and Educational Services available by special arrangement.

LMI ***Laboratory Microsystems Incorporated***
Post Office Box 10430, Marina del Rey, CA 90295
Phone credit card orders to: ***(213) 306-7412***

Overseas Distributors.
Germany: Forth-Systeme Angelika Flesch, Titisee-Neustadt, 7651-1665
UK: System Science Ltd., London, 01-248 0962
France: Micro-Sigma S.A.R.L., Paris, (1) 42.65.95.16
Japan: Southern Pacific Ltd., Yokohama, 045-314-9514
Australia: Wave-onic Associates, Wilson, W.A., (09) 451-2946

Number **63** on the Reader Service Card

ing actual jump times on the 286 appears at first cut to be black magic). Thanks to changes induced in the prefetch queue by the timing routines, results aren't necessarily accurate to a cycle, but then there's no way around that with a software solution. On quick examination, it looks to me as if Sheppard's done an excellent job.

Reader Feedback

Tom Heavey provided a solution to my problem of how to generate code that assembles at a specific address (for ROM, in my case). Tom's solution bypasses the Macro Assembler entirely; he suggests using a program called Link and Locate, from Systems and Software, Inc. ((714) 241-8650), in place of the standard MS DOS linker, with which it is compatible. According to Tom, Link and Locate adds a locate phase after the link phase, during which you can specify exactly where segments are to go.

Charles Quinton also provided a solution, but one that I can get to work only under certain circumstances. Charles points out that it's acceptable to have two relocatable values in an expression, so long as they combine to generate an absolute, so he suggests using

```
DB      2000h - ( $ - ProgramStart ) dup (0)
```

where ProgramStart is the offset of the first byte of the program. The assembler does indeed accept this syntax, and so long as ProgramStart is in the same module as the above DB statement, everything's fine. If, however, ProgramStart is in another module, too many bytes are inserted because "$" is relative to the start of the module, not to the start of the final linked program. Perhaps Charles meant this technique to be used only in single-module programs, or perhaps I'm missing something. Charles, can you write and let us know?

John Navas II provided a similar solution. Instead of the start of the program, however, he subtracts the segment name from the location counter. So far as I can tell, this operates exactly as Charles Quinton's approach does. John also sent in answers to two other MASM problems I've discussed here; they arrived too late to make it into this article, but I'll run them next time.

Thanks to all of you.

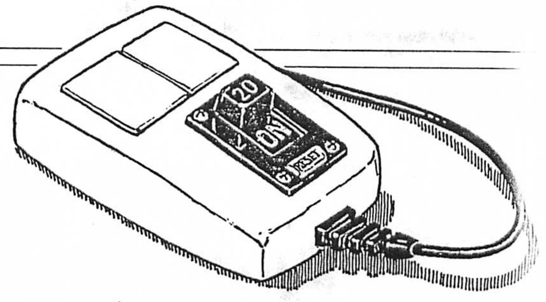
Next Time

We've almost completed our tour of GDC data flow—only the set/reset features remain. I'll cover these odd but useful features next time. If there's room, I'll begin to strike out into the more specialized read and write modes of the EGA.

See you then. □

Michael Abrash is a Senior Software Engineer for Orion Instruments of Redwood City, CA, a manufacturer of PC-based instrumentation and microprocessor development systems.

Code follows.



EGA Set/Reset Capabilities

by
Michael Abrash

This month I'll examine a powerful but confusing aspect of the EGA: the set/reset circuitry. I'll also discuss a few odds and ends and print reader responses to some of the questions I've posed in previous articles.

The Set/Reset Circuitry

At last we come to the final aspect of data flow through the Graphics Controller (GC) on write mode 0 writes, the set/reset circuitry. Figure 1 shows data flow on a write mode 0 write. The only difference between this figure and the similar figure in the last article is that on its way to each plane the rotated CPU data passes through the set/reset circuitry, which may or may not replace the CPU data with set/reset data. Briefly put, the set/reset circuitry enables the programmer to elect to independently replace the CPU data for any or all planes with either 00 or 0FFH.

What is the use of such a feature? Well, the standard way to control color is to set the Map Mask register to enable writes to only those planes that need to be set to produce the desired color. For example, the Map Mask register would be set to 09H to draw in high-intensity blue; here, bits 0 and 3 are set to 1, so only the blue plane (plane 0) and the intensity plane (plane 3) are written to.

Remember, though, that planes that are disabled by the Map Mask register are not written to or modified in any way. This means that the approach above works only if the memory being written to is zeroed; if, however, the memory already contains non-zero data, that data will remain in the planes disabled by the Map Mask, and the end result will be that some planes contain the data just written and other planes contain old data. In short, color control using the Map Mask does not force all planes to contain the desired color; in particular, it is not possible to force some planes to zero and other planes to one in a single write with the Map Mask register.

The program in Listing 1 illustrates this problem. A green pattern (plane 1 set to 1, planes 0, 2 and 3 set to 0) is first written to display memory. Display memory is then filled with blue (only plane 0 set to 1), with a Map Mask setting of 01H. Where the blue crosses the green, cyan is produced, rather than blue, because the Map Mask register setting of 01H that produces blue leaves the green plane (plane 1) unchanged. In order to generate blue unconditionally, it would be necessary to set the Map Mask register to 0FH, clear memory, and then set the Map Mask register to 01H and fill with blue.

The set/reset circuitry can be used to force some planes to zero and others to one during a single write and so provides an efficient way to set all planes to a desired color. The set/reset circuitry works as follows:

For each of the bits 0 - 3 in the Enable Set/Reset register (Graphics Controller register 1) that is 1, the corresponding bit

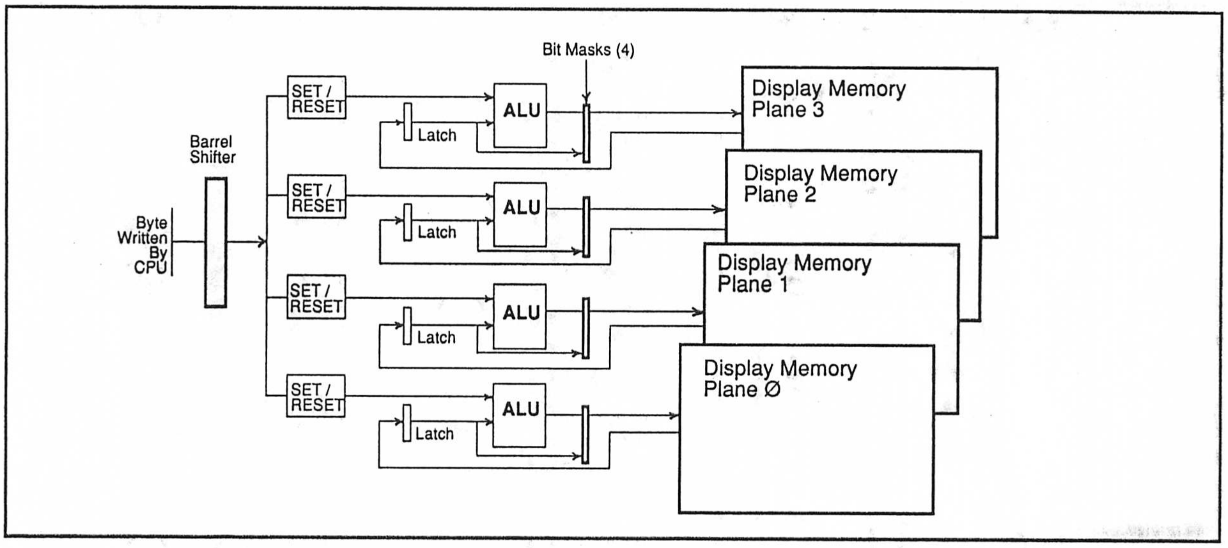


Figure 1: Data flow through the Graphics Controller chips on write mode

in the Set/Reset register (GC register 0) is extended to a byte (0 or 0FFH) and replaces the CPU data for the corresponding plane. For each of the bits in the Enable Set/Reset register that is 0, the CPU data is used unchanged for that plane (normal operation). For example, if the Enable Set/Reset register is set to 01H and the Set/Reset register is set to 05H, the CPU data is replaced for plane 0 only (the blue plane), and the value it is replaced with is 0FFH (bit 0 of the Set/Reset register extended to a byte). Note that in this case, bits 1 - 3 of the Set/Reset register have no effect.

It is important to understand that the set/reset circuitry directly replaces CPU data in Graphics Controller data flow. Refer again to Figure 1 to see that the output of the set/reset circuitry passes through (and may be transformed by) the ALU and the bit mask before being written to memory, and even then the Map Mask register must enable the write. When using set/reset, it is generally desirable to set the Map Mask register to enable all planes the set/reset circuitry is controlling since those memory planes that are disabled by the Map Mask register cannot be modified, and the purpose of enabling set/reset for a plane is to force that plane to be set by the set/reset circuitry.

Listing 2 illustrates the use of set/reset to force a specific color to be written. This program is the same as that of Listing 1 except that set/reset rather than the Map Mask register is used to control color. The preexisting pattern is completely overwritten this time because the set/reset circuitry writes zeroes to planes that must be off as well as ones to planes that must be on.

Listing 3 illustrates the use of set/reset to control only some, rather than all, planes. Here, the set/reset circuitry forces plane 2 to 1 and planes 0 and 3 to 0. Because bit 1 of the Enable Set/Reset register is 0, however, set/reset does not affect plane 1; the CPU data goes unchanged to the plane 1 ALU. Consequently, the CPU data can be used to control the value written to plane 1. Given the settings of the other three planes, this means that each bit of CPU data that is 1 generates a brown pixel and each bit that is 0 generates a red pixel. Writing alternating bytes of 07H and 0E0H, then, creates a vertically striped pattern of brown and red.

In Listing 3, note that the vertical bars are 10 and 6 bytes wide and do not start on byte boundaries. Although set/reset replaces an entire byte of CPU data for a plane, the combination of set/reset for some planes and CPU data for other planes, as in the example above, can be used to control individual pixels.

There is no clearly defined role for the set/reset circuitry as there is for, say, the bit mask. In many cases, set/reset is largely interchangeable with CPU data, particularly with CPU data written in write mode 2 (write mode 2 operates similarly to the set/reset circuitry). The most powerful use of set/reset, in my experience, is in applications such as the example of Listing 3, where it is used to force the value written to certain planes while the CPU data is written to other planes.

Notes on Set/Reset

The set/reset circuitry is not active in write modes 1 or 2. An explanation of those two write modes is a topic for another article.

Be aware that because set/reset directly replaces CPU data, it does not necessarily have to force an entire display memory byte to 0 or 0FFH, even when set/reset is replacing CPU data for all planes. For example, if the Bit Mask register is set to 80H, the set/reset circuitry can modify only bit 7 of the destination byte in each plane since the other seven bits will come from the latches for each plane. Similarly, the set/reset value for each plane can be modified by that plane's ALU.

An Update

In an earlier article, I suggested the use of word OUTs (OUT DX,AX) to set indexed EGA registers. I have since been informed that this does not work in some early models of AT&T personal computers. If your software is likely to run in such a computer, I suggest you not use word OUTs. One good habit to get into is the use of macros or subroutine calls to set indexed registers; this practice makes it easy to convert from byte OUTs to word OUTs, or vice versa, if needed.

Reference Material

Several readers have asked where they can get information about programming the Color Graphics Adapter and Hercules Graphics Card. The *IBM Technical Reference, Options and Adapters* manual is a reference for the CGA and Monochrome Display Adapter, as well as the primary reference for the EGA. The best reference for the CGA, however, is "The IBM Color/Graphics Adapter" by Thomas V. Hoffmann, *PC Tech Journal*, Vol. 1, No. 1, July, 1983.

The primary reference for programming the HGC is the manual Hercules supplies with the board. Most companies that supply Hercules-compatible boards also provide programming information.

Reader Feedback

In my last article, I included responses from several readers explaining how to generate ROMable code at a specific address. This month's reader responses include answers to all three MASM questions I've posed and what surely is the definitive word on the problems of generating code at a specific address.

John Navas II answered all three MASM questions, as follows. The sample program to which he refers is shown in Listing 4.

> My solutions to your three puzzles range from crude to elegant to simple. I've enclosed a sample program that demonstrates all of them.
>
> (0) You are correct that MASM has only a single symbol space so that you cannot normally reuse symbols in different STRUCtures. However, if, for example, you are defining local STRUCtures for small procedures, there is a crude way to reuse symbols. Simply enclose an entire procedure including any local STRUCtures in a MACRO and declare all of the symbols LOCAL. When the macro is expanded, the assembler will generate unique symbols automatically, and there is no danger of referencing the wrong STRUCture. If you reuse the same MACRO name over and over, MASM shouldn't run out of memory. The downside is that the assembly takes longer, the listing is harder to read, and brackets must be used with LOCAL symbols instead of the more logical (.) STRUCture field-name operator (at least under MASM Version 4.0).
>
> (1) Using STRUCtures to define negative offsets from the BP register in a stack frame (typically for local dynamic storage allocation) requires the SIZE of the STRUCture to be subtracted from BP (for example, [bp-SIZE locs].field1).

The EQU directive can make it all quite elegant. First EQUate a symbol to the SIZE of the STRUCture, adjusted if necessary to an even number. Then EQUate a second symbol to BP minus the first symbol, all enclosed in brackets. Subtract the first symbol from SP to allocate storage. Use the second symbol, followed by the STRUCture field name either separated by the (.) STRUCture field-name operator or enclosed in brackets, to access individual fields (for example, loc.w1 or loc[field1])).

(2) As you noted, the count for a DUP operator must be a constant expression. Since the location-counter is relocatable, it must be converted to a constant value before it can be used in a DUP count. The simplest way to convert it is to subtract the SEGMENT name. Your example might become:

```
    DB      (2000h-($-segname)) DUP( 0 )
```

Note that the inner parentheses are necessary to force MASM to first perform the necessary conversion and that parentheses must of course enclose the DUP value.

Thank you, John. I must repeat a comment from the last article: The last-mentioned solution works only within the first module linked, which makes sense since the whole point of your solution is to generate an absolute value at assembly time and relocation takes place at link time.

Charles Clinton wrote at considerable length about the problem of putting code or data at an absolute offset, shedding light on the somewhat bewildering operation of MASM and proving both that there's a great deal for a patient and persistent reader to learn from the MASM manual and that there are mysteries that elude even the most patient and persistent reader.

Charles' letter follows. The letter was in the form of a letter to the editor; hence, the reference to the author in the third person. Please note that apart from correcting typos, I present the letter exactly as Charles wrote it; all editorial comments are his.

The difficulty that Michael Abrash is encountering in attempting to convince MASM to assemble code up to a certain location (*PJ* 5.1, p. 42) is that MASM has no idea where the code that it is currently assembling is going to end up. The code is assembled into the current program segment, by which MASM means "a collection of instructions and/or data whose address are all relative to the same segment register" (MASM 4.0 Ref. Man., Sect. 3.4, p. 27) and not relative to an absolute memory address or offset.

The solution to the problem involves making the assumption that since we know where the code is going to go, we can talk about absolute memory locations by describing them relative to the code.

The section on arithmetic operators (Sect. 5.3.1, p. 78) says that all arithmetic expressions must either be integers or, in some special cases, "relocatable memory operands." The section on relocatable [memory] operands (Sect. 5.2.3, p. 69) defines them as "any symbol that represents the memory address. . . of an instruction or of data. . . and [has] no explicit value until the program has been linked." Examining the definition of the current location counter (Sect. 5.2.4, p. 69), we see that it has "the same attributes" as a "near label." Flipping around some more, we find the section on symbol declarations (Sect. 4.4, p. 54) that implies that a label generates a symbol. This means that the $ is a relocatable memory operand (if our assumptions have not gone awry).

Going back to the section on arithmetic operators, we find that we can add or subtract a constant from a relocatable memory operand or we can find the absolute difference between two memory operands in the same [assembler] segment. We can exploit the difference operation to find our location in memory. If we know that a particular label is going to be at a particular location in memory and we know how to calculate the distance between two labels, we can calculate the location of a second label.

For example, with the problem that Mr. Abrash is presenting we need to determine the location of the end of the assembled code so that we can zero fill to the end of the ROM chip. If we define a label at the start of the ROM chip, like this:

```
code        segment     'ROM'
            assume      cs: code
            org         0
  rom_start:
            ... code ...
```

we can then calculate how many bytes have been assembled, thusly:

```
number_assbld = $ - offset rom_start
```

The number of bytes that need to be filled is the length of the ROM chip (2000H) less the number of bytes assembled:

```
to_fill = 2000H - number_assbld
```

The statement that does the filling is then:

```
    db      to_fill dup (0)
```

Putting it all together, and simplifying the expressions, yields:

```
code            segment     'ROM'
                assume      cs: code
                org         0
rom_start:
        ... code ...
            db              2000H - ($ - offset rom_start) dup (0
code    ends
        end
```

The key to this solution is remembering that MASM will maintain that it does not know in what manner the linker will manipulate the segment being assembled. In fact, in most cases the linker will concatenate many code and data segments and then concatenate the segments into groups (which are not necessarily contiguous) and classes. This flexibility and power is what gives one the ability to mix multiple languages and memory models in a single program and allow exact specification of the ordering and arrangement of memory. Unfortunately, it has the side effect of making some of the simplest tasks complicated.

The seemingly obvious solution to this problem of using the "at address" combine clause in the segment declaration will not work as Microsoft's intended use of this was to allow the user to describe existing data areas, and not define them. So they did not make the expression evaluator smart enough to recognize that the current location counter ("$") can sometimes represent a fixed address.

A subtler problem seems to exist in the offset operator. At first glance, it would seem that

```
    db      2000h - offset $ dup (0)
```

should produce the desired effect. Although the MASM manual claims that the offset operator returns the number of bytes between the symbol and the beginning of its segment (Sect. 5.3.13, p. 88), a little experimenting shows that this is not entirely true. The .TYPE operand (Sect. 5.3.15, p. 89) reveals some curious things about the way MASM represents numbers. Assembling the following code reveals the problem:

```
code        segment     'ROM'
            assume      cs: code
            org         10
a           =           10
atype       =           .type a
b           =           offset $
btype       =           .type b
code        ends
            end
```

Software Tools
For Programmers & Non-Programmers

Get 'State of the Art' performance and save valuable time with these high quality utilities!

Opt-Tech Sort™

Opt-Tech Sort is a high performance Sort/Merge/Select utility. It can read, sort and write a file faster than most programs can even read the data. Example: 1,000 records of 80 bytes can be read, sorted and a new file written in less than 10 seconds (IBM XT). Opt-Tech Sort can be used as a stand-alone program or called as a subroutine to over 25 different programming languages.

All the sorting, record selection and reformatting facilities you need are included. A partial list of features includes: The ability to process files of any size. Numerous filetypes are supported including Sequential, Random, Delimited, Btrieve, dBASE II & III and many others. Up to 10 key fields can be specified (ascending or descending order). Over 16 different types of data supported. Powerful record selection capability allows you to specify which records are to be included on your output. Record reformatting allows you to change the structure of your output record and to output special fields such as record numbers for use as indexes.

MS-DOS $149. ★ NEW ★ Xenix $249.

★ NEW ★ VERSION **On-Line Help™** ★ NEW ★ VERSION

On-Line Help allows you to easily add "Help Windows" to all your programs. On-Line Help is actually two help packages in one. You get BOTH Resident (pop-up) and Callable Help Systems.

The resident version allows you to add help to any system. Your Help System is activated when the "Hot Keys" that you specify are pressed. You can then chain between help windows in any manner you desire.

The callable version allows you to easily display help windows from your programs. A simple call to the help system makes the window appear. The original screen is automatically restored when the help window is cleared. On-Line Help is callable from over 20 different languages.

You have full control over the help window content, size, color and location.

MS-DOS $149. **Demo $10.** (apply toward purchase).

Scroll & Recall™

Scroll & Recall is a resident screen and keyboard enhancement. It allows you to conveniently scroll back through data that has gone off the top of your display screen. Up to 27 screens of data can be recalled or written to a disk file (great for documenting systems operations). Also allows you to easily recall and edit your previously entered DOS commands without retyping.

Scroll & Recall is very easy to use. It's a resident utility that's always there when you need it. MS-DOS $69.

Visa, M/C, AMEX, Check, Money Order, COD or Purchase Orders accepted.

To order or to receive additional information just call and receive immediate highly qualified attention!

Opt-Tech Data Processing
P.O. Box 678 — Zephyr Cove, NV 89448
(702) 588-3737

Number 16 on the Reader Service Card

Upon examining the listing, we see:

```
          N a m e                          Type          Value     At

A . . . . . . . . . . . . . .             Number        000A
ATYPE . . . . . . . . . . . .             Number        0020

B . . . . . . . . . . . . . .             Number        000A      CO
BTYPE . . . . . . . . . . . .             Number        0022
```

By the MASM definition of OFFSET, we would expect A and B to have identical values and both be plain numbers. Rather, what we see is that OFFSET is failing to strip off the attribute of 'CODE' from the expression "OFFSET $." Attributes are a little tricky in MASM as they are not defined in the MASM manual. They are not in the index, and the only mention of attributes that I can find are in the sections "Location-Counter Operand" (Sect. 5.2.4, p. 69) and ".TYPE Operator" (Sect. 5.3.15, p. 89), both of which I consider cryptic descriptions at best. There does not seem to be an operator for changing the attribute of an expression, which is what is called for here.

Thank you, Mr. Clinton, for guiding us deep into the murky depths of MASM as it pertains to the problem of generating code or data at an absolute offset. As has been my experience with MASM, there is indeed an explanation and a workaround, but it sure seems harder than need be to perform what should be a straightforward task. ☐

Code follows.

EVERLOCK COPY PROTECTION

Designed for user-transparency, clone compatibility & strength. It features:

- no I/O plugs or special media
- FULL hard disk & cartridge support
- file-server network support
- variable number of installs (0-99)
- demo diskette option with unlock
- protected upgrades by modem/BBS

All this for **$495** with no meter counts.
Free info & demo disk.
(Duplication services also available).

AZ

AZ-Tech Software, Inc.
305 E. Franklin, #A4A
Richmond, MO 64085
(816) 776-8153

Number 51 on the Reader Service Card

Write Mode 3 of the VGA

by
Michael Abrash

Well, the VGA is here, and it's a dandy, with 256 color capability (from 256K, no less), higher resolution, square pixels, readable registers (finally!), lots of new BIOS functions, better text, relatively good performance, and more. The VGA represents a whole new level of video capability, providing all the basics needed for decent business graphics for the first time in standard IBM graphics. The VGA provides reasonable backward compatibility with the CGA, MDA, and, most importantly, EGA. In short, the first video circuitry IBM has ever built into a motherboard is good, about as good as it could be given the lack of a built-in graphics processor and the constraints of backward compatibility.

For *PJ* readers, of course, the concern is how the VGA will affect their software development efforts. The answer: a great deal. All Micro Channel computers have the VGA built in; the PC/XT/AT version of the VGA, the Display Adapter, is priced at a fairly inexpensive $595; and third-party video companies are scrambling to be the first to clone the VGA. Between the benefits of the VGA and IBM's commitment to the new standard, the VGA should take off faster than even the EGA did.

So, readers, support the VGA in your development efforts—it's the hottest thing around right now. Besides, if you want your software to run in the Model 50 and up PS/2 computers, it *has* to run on the VGA, although the BIOS does a good job of masking the differences between the VGA and earlier adapters for older applications.

I'm not going to do a review/evaluation of the VGA; you'll have plenty of opportunities to read such articles in other magazines, and my theory is that *PJ* readers want practical tips and working code. With that in mind, I'm going to note a few points about identifying and programming the VGA and then revisit an old application in order to illuminate an unusual VGA programming feature, write mode 3.

Notes on Programming the VGA

The first point about the VGA is that while it's certainly different from earlier adapters, most old software should run on it. CGA and MDA software should run on the VGA to about the extent to which they run on the EGA, which is to say the VGA is BIOS-level compatible with the CGA and MDA. Most EGA software should run on the VGA so long as the BIOS is used whenever possible. While changing the hardware itself considerably, IBM managed to retain in the VGA the combination BIOS/register interface through which well-behaved application programs access the EGA. This doesn't mean that programs that load all the registers directly, or, indeed, touch any more registers than they absolutely must, will run; most VGA registers are, in fact, not EGA compatible. The EGA programming techniques I've discussed in *PJ* will work on the VGA. The rule of thumb: When in doubt, refer to the EGA and VGA technical reference material and make sure that the registers you're accessing are functionally the same.

VGA Reference Material

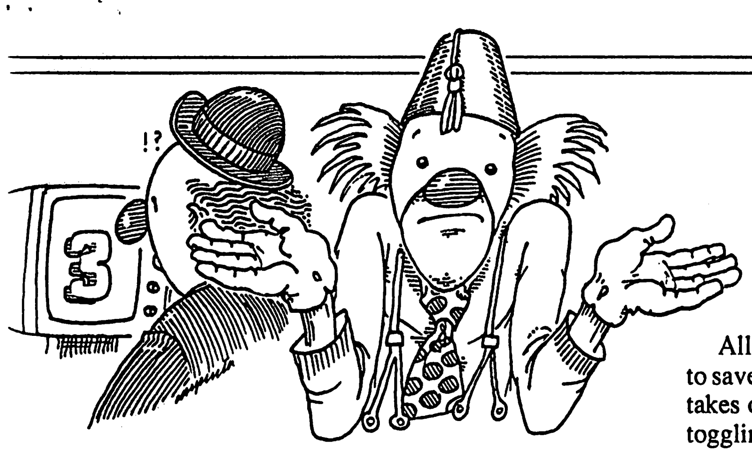
Which brings me to my next point: The VGA technical reference material is included in the *Technical Reference* manuals for the Micro Channel computers, including the *Model 50 and 60 Technical Reference* and the *Model 80 Technical Reference*. The Display Adapter technical reference material is available as a separately purchased supplement, the *Display Adapter Technical Reference* (part number S68X-2251-0). Oddly enough, the register descriptions in the *Display Adapter Technical Reference* and in the technical references for the Micro Channel computers are not identical although they seem to describe the same registers. It may be worth your while to buy both the *Display Adapter Technical Reference* and one of the Micro Channel computer technical references in order to have two descriptions of some registers.

The *Display Adapter Technical Reference* includes a specification of the interface to the Display Adapter BIOS (which seems to be functionally pretty much the same as the VGA BIOS) as well as a register-level description of the Display Adapter. The technical references for Micro Channel computers do not contain any BIOS interface information; if you don't have the *Display Adapter Technical Reference*, you can get a specification of the entire IBM BIOS interface (including all IBM video BIOS functions) by purchasing IBM's *BIOS Interface Technical Reference* manual (part number S68X-2260-00).

Differences between the VGA and the Display Adapter

IBM notes a few significant differences between the Display Adapter and the VGA. (Incidentally, VGA stands for Video Graphics Array, which describes the chip on the Micro Channel motherboard. So far as I can determine, the same chip is on the Display Adapter—at least it has the same part number and does the same things—but it's easiest to refer to the motherboard video circuitry as the VGA and to the PC/XT/AT plug-in board as the Display Adapter.) I'll discuss these differences, which include vertical interrupt capability and monochrome text attributes, next.

Although the VGA can generate vertical interrupts of the same sort as the EGA does, the Display Adapter has no vertical interrupt capability. The reasoning behind this difference isn't clear, but I tested the Display Adapter and the vertical interrupt is indeed not available.



On page 4-5 of the *Display Adapter Technical Reference*, there is a note to the effect that "the Personal System/2 Display Adapter displays a reverse video intensified character as white on white." From that, it's not clear exactly what the problem is, so I did some experimentation. So far as I can tell, when the Display Adapter is in mode 7, characters for which either the foreground or background attribute is 8 are displayed as white-on-white and are consequently not readable. The VGA, on the other hand, does display such text correctly. I'm not sure why this problem occurs or what a workaround might be; anyone out there have any suggestions?

By the way, the VGA is capable of driving either a color or a monochrome monitor automatically. (The monitors must be analog monitors compatible with IBM's PS/2 monitors, the 8503, 8512, 8513, and 8514.) There are no switches; the VGA IDs the monitor through the video connector. Color summing to gray scales is performed when driving a monochrome monitor. What's more, in Micro Channel computers, programs can select any mode directly through INT 10h, function 0; for example, a program can go from mode 3 (color text) to mode 7 (monochrome text) simply by performing

```
MOV AX,7
INT 10H
```

It is no longer necessary to fiddle with the equipment flag in order to switch between color and monochrome modes (in fact, the VGA BIOS changes the equipment flag as needed on mode sets), and the VGA by itself can act like any of the CGA, MDA, or EGA adapters at any time (although it can do this only serially; it can't emulate both at once since it is in truth only a single adapter).

The Display Adapter can drive either a monochrome or a color monitor and also features switchless installation. Switching between color and monochrome operation, however, still requires changing the equipment flag; unlike the VGA, the Display Adapter is constrained by the possible presence of other adapters in the system. Overall, IBM seems to have designed the BIOS and the VGA in such a way that application programs will run just as they would on earlier adapters in most cases.

One more note about programming the VGA: Don't use word OUTs to the Attribute Controller at port 3C0h. The EGA addresses the Attribute Controller at both 3C0h and 3C1h, so word OUTs to the Attribute Controller work despite its strange toggling nature. (See my article "Inside the EGA," *PJ*, Volume 5.1, Jan/Feb 1987, for a description of programming the Attribute Controller.) The VGA addresses the Attribute Controller at 3C0h only, meaning that the high byte of a value written to the VGA's Attribute Controller by a word OUT goes into the bit bucket. Word OUTs to the Sequencer, the CRT Controller, and the Graphics Controller seem to work fine on the VGA.

All VGA registers are readable, which makes it possible to save the video context (for example, in a TSR program that takes over the screen). While the VGA maintains the EGA's toggling Attribute Controller port operation on writes, it implements Attribute Controller reads in a sensible fashion. The Attribute Controller Index register is always readable at 3C0h, and the Attribute Controller Data register is always readable at 3C1h. Sadly, the need for EGA compatibility prevented IBM from making Attribute Controller writes operate equally sensibly.

There is one aspect of the VGA state that can't be read out and which consequently can't be saved on a context switch, and that's the state of the Attribute Controller index/data toggle. As a result, it seems advisable to disable interrupts while programming Attribute Controller registers, lest an interrupt come along and trigger a TSR that accesses the Attribute Controller. Better yet, use BIOS function to program Attribute Controller registers whenever possible. The palette RAM registers (Attribute Controller registers 0 through 0Fh) can be written (and read) through the BIOS, and these are usually the only Attribute Controller registers that application programs change.

Identifying the VGA

IBM has added a couple of functions to the VGA BIOS (and to the Model 30 BIOS as well) that make it much easier to identify installed adapters and their capabilities. Both are accessed through the standard INT 10h video interrupt.

Function 1Ah returns a display combination code indicating the type of the currently active display adapter and the alternate display adapter, if any. Function 1Ah isn't supported by earlier adapters, so the first test is whether function 1Ah is supported by the installed adapter. If, on return from a function 1Ah call, register AL is set to 1Ah, then function 1Ah is supported. If the function is supported, BL returns the active display code and BH returns the alternate display code. Function 1Ah can also be used to change the active and alternate display adapters.

Function 1Bh returns a buffer containing functionality/state information. This information includes the current display mode, the width of the screen, the display buffer size, other BIOS variables, the display combination code, the number of colors supported by the current mode, and font information. Best of all, it includes information about the video modes supported by the active adapter. I haven't had a chance to dig into the functionality/state information yet, but it looks like a gold mine. If this function had been available since the MDA and CGA, the world of IBM graphics would be a lot less confusing.

Write Mode 3

The MDA offered only text mode, so it was easy to program. The CGA had a couple of graphics modes, and while the bit map organization in these modes was somewhat complex, at least the bit map was directly addressed as one linear block of memory. The EGA added the complications of three write modes and two read modes, all across four planes. The VGA supports all the modes of the earlier adapters and adds one new write mode to the list, write mode 3.

Write mode 3 is strange indeed, and its use is not immediately obvious. In write mode 3, set /reset is automatically enabled for all four planes (the Enable Set/Reset register is ignored). The CPU data byte is rotated and then ANDed with the contents of the Bit Mask register, and the result of this operation is used as the contents of the Bit Mask register alone would normally be used. (If this is *Greek* to you, I suggest you read my earlier four articles on graphics in *PJ*, Volumes 5.1 through 5.4. There's no way to understand write mode 3 without understanding write mode 0 first.)

That's what write mode 3 does—but what is it for? I'm not certain what IBM had in mind, but the best applications for write mode 3 I've been able to think of to date have to do with bit-mapped text applications. Write mode 3 seems reasonably well-suited to drawing large quantities of graphics-mode text quickly without wiping out the background in the process.

Listing 1 is a modification of the code presented in my article "EGA Data Control" (*PJ*, Volume 5.3, May/June 1987). That code used the data rotate and bit mask features of the EGA to draw bit-mapped text in write mode 0. Listing 1 uses write mode 3 to draw bit-mapped text, and in the process gains the important benefit of preserving the background into which the text is being drawn. Where the original text-drawing code drew the entire character box for each character, with 0 bits in the font pattern causing a black box to appear around each character, the code in Listing 1 affects display memory only when 1 bits in the font pattern are drawn. As a result, the characters appear to be painted into the background, rather than over it. Another advantage of the code in Listing 1 is that the characters can be drawn in any of the 16 available colors.

The key to understanding Listing 1 is understanding the effect of ANDing the rotated CPU data with the contents of the Bit Mask register. The CPU data is the pattern for the character to be drawn, with bits equal to 1 indicating where character pixels are to appear. The Data Rotate register is set to rotate the CPU data to pixel-align it since without rotation characters could be drawn only on byte boundaries. At the same time, the Bit Mask register is set to allow the CPU to modify only that portion of the display memory byte accessed that the pixel-aligned character falls in so that other characters and/or graphics data won't be wiped out. The result of ANDing the rotated CPU data byte and the contents of the Bit Mask register is a bit mask that allows only the bits equal to 1 in the original character pattern (rotated and masked to provide pixel alignment) to be modified by the CPU; all other bits come straight from the latches. The latches should have previously been loaded from the target address, so the effect of the ultimate bit mask value is to allow the CPU to modify only those pixels in display memory that correspond to the bits equal to 1 in that part of the pixel-aligned character that falls in the currently addressed byte. The color of the pixels set by the CPU is determined by the contents of the Set/Reset register.

Whew. It sounds complex, but given an understanding of what the data rotator, set /reset, and the bit mask do, it's not that bad. One good way to make sense of it is to refer to the original text-drawing program in *PJ* 5.3, and then see how Listing 1 differs from that program.

LEARN about . . .

Neural Nets

A major upheaval in thinking about artificial intelligence is underway. Interest is accelerating in the "neural net" approach, which is loosely based on human brain research. A tutorial introduction to this exciting new technology is now available in a model called Netwurkz (tm).

Netwurkz eliminates the complicated math found in "unsupervised learning" so that underlying principles may be grasped quickly. The network "learns" when you add new DATA statements (interconnects), or modify existing ones. A trace on/off feature prints the neural net computation in progress. List based internal representation yields speed.

Netwurkz is intended for people not familiar with neural modeling who want to maintain or broaden their technical skills and background. Source for all neural net software and an object-only copy of the PL/D compiler are included

Compilers

PL/D (tm) is a new system language that contains documentation so definitive that your understanding of all languages is enhanced. Like other languages, PL/D is described in narrative form, alphabetically by token and via flow diagrams. However, PL/D also comes with self-compiling source! If you have a question after reading the manual, you can look at the source. If you do not accept the answer, you can change it by modifying the code.

What you learn from PL/D also applies to all compilers: symbol table organization, macro handling, nested structure implementation, cross reference listing, code generation, etc. You will study an unusually efficient implementation: PL/D is 2-3 times faster than the recent "fast" C compilers.

If you are a bright self-sufficient type who wants to master your development tools, PL/D is for you.

PC or compatible with 256K and DOS 2.0 or above and 5.25 inch diskette required.

master card

VISA

DAIR Computer Systems 3440 Kenneth Drive, Palo Alto CA 94303
(415) 494-7081

Name:	
Company:	
Address:	
City, State, Zip:	
Phone:	
VISA, M/C:	Expire:
Netwurkz: neural net demo	$79.95
PL/D: fast self-compiles	$124.95
Both of above	$154.95
Sales Tax, CA only	
Ship overseas	$10.00
TOTAL	

Want to know more?
(415) 494-7081

Available Now

Number 5 on the Reader Service Card

Software Developers

We need your program!

Do you have a program that's good enough to sell, but don't want the problems or financial risk of producing, typesetting, printing, packaging, warehousing, marketing, distributing and supporting a product?

Why start your own software house?
We've done it for you!

Merlin Publishing Group
is now accepting submissions of micro-computer software for publication.

You get:
- initial cash payments
- generous royalties
- to spend your time programming

But we can't help you if you don't submit. Call or write today for our submission guideline kit.

MERLIN
PUBLISHING GROUP ™

1240 Johnsons Ferry Place, Suite A10
Marietta, GA 30068
(404) 977-6034

P. S. See us in Atlanta at COMDEX-Spring '88

Number 141 on the Reader Service Card

It's worth noting that the same effect generated by Listing 1 could have been accomplished without write mode 3, and at relatively little performance cost. Write mode 0 could have been used instead. Rather than let write mode 3 rotate the CPU data and AND it with the contents of the Bit Mask register, the CPU could simply have rotated the CPU data directly and ANDed it with the value destined for the Bit Mask register and then set the Bit Mask register to the resulting value. Additionally, set/reset could simply be enabled for all planes, emulating what write mode 3 does to provide pixel colors. While this arrangement would have required a bit more bit manipulation by the CPU and would have required CPU rotation, which is slower than the VGA's rotator, it would have saved setting the Data Rotate register and actually might even have been a faster routine overall.

Write mode 3 could be considerably more useful for drawing large blocks of text. For example, suppose that we were to draw a line of 8-dot-wide, bit-mapped text 40 characters long. It would be very possible to set up the bit mask and data rotation as appropriate for the left portion of each bit-aligned character (the portion of each character to the left of the byte boundary) and then draw the left portions only of all 40 characters in write mode 3. Then the bit mask could be set up for the right portion of each character, and the right portions of all 40 characters could be drawn. The VGA's fast rotator would be used to do all rotation, and the only OUTs required would be those required to set the bit mask and data rotation. This technique could well outperform single-character bit-mapped text drivers such as the one in Listing 1 by a significant margin. Listing 2 illustrates one implementation of such an approach. Incidentally, note the use of the 8 x 14 ROM font in Listing 2, rather than the 8 x 8 ROM font used in Listing 1. There is also an 8 x 16 font stored in ROM, along with the tables used to alter the 8 x 14 and 8 x 16 ROM fonts into 9 x 14 and 9 x 16 fonts.

It's likely that there's a better application for write mode 3 than the sort of bit-mapped text handling I've discussed. If you think of such an application, please write and let *PJ*'s readers know about it.

Preserving Reserved Bits

Note that the code in Listing 1 uses the readable register feature of the VGA to preserve reserved bits and bits other than those being modified. The EGA has no readable registers, and so it was necessary to set all bits in a register whenever that register was modified. The VGA makes it possible to change only those bits of immediate interest, and this procedure is highly recommended since IBM (or clone manufacturers) may well use some of those reserved bits in the future.

In Conclusion

The VGA is a complex, powerful, fascinating video standard. There's a heck of a lot to learn, and knowledge is the key to high-performance VGA programs. One area I'll make it a point to cover soon is 256 color mode, another is the ability to select colors from a set of 256K, and yet another is text mode font handling. And then there's the new two-color high-resolution mode, the pel panning compatibility feature, the high-bandwidth blanking bit, the . . . . Yes indeed, there's a lot to cover next time. See you then. ♦

Michael Abrash is an Engineering Fellow working on advanced graphics projects for Video Seven, Inc., in Fremont, CA.

Code follows.

NEW!
FASTER THAN EVER!
DeSmet C V3.0

FASTER C DEVELOPMENT

Invoke the DeSmet C compiler from the SEE™ full screen editor and the first error will return you immediately to SEE at the error line with the error message displayed.

FASTER COMPILATION

When you don't use inline assembly code or don't want to see the ASM88 output, the V3.0 compiler produces object code directly - making DeSmet C up to *twice as fast* as before.

PLUS EXPANDED STANDARD LIBRARY

Networking, path, file, time, enhanced string functions, environment support now included.

FULL FEATURES WITH EVERY PACKAGE
—— ONLY $109 ——

C Compiler, Assembler, Binder, Librarian, Execution Profiler, Overlays, 8087 and S/W Floating Point, Full STDIO Library and Full Screen Editor (SEE). Debugger and Large Case options available at $50 each.

C Ware Corporation
P.O. Box 428, Paso Robles, CA 93447
Phone: (805) 239-4620 Telex: 358185 BBS: (805) 239-4834
We accept VISA, MC & AMEX. Call now and we'll ship today.
Street Address: 945 Spring #14, Paso Robles, CA 93446

Number 38 on the Reader Service Card

Introducing
The only code-intensive publication on C for MS-DOS® systems.

• Code-intensive: pages of clear, fully-commented and portable C. Extensive textual support for the code.
• In-depth coverage of algorithms: detailed explanations presented from different perspectives by several authors.

Call or write today for your subscription.

1 year (4 issues) $21. 2 years (8 issues) $38.

The C Gazette

1341 Ocean Ave. #257 Santa Monica, CA 90401
(213) 473-7414

Number 144 on the Reader Service Card

Yet Another VGA Write Mode

by
Michael Abrash

Installment 6: In which we venture ever deeper into unknown waters, encountering yet another of those damnable write modes.

In the last installment of *On Graphics*, we learned about the markedly peculiar write mode 3 of the VGA—after having spent four installments learning the ins and outs of the EGA's write mode 0, touching on write mode 1 as well. Happily, it turns out that the VGA supports all the write modes (write modes 0, 1, and 2) of the EGA and also supports the same read modes as the EGA. Which leaves two burning questions: What is write mode 2, and how the heck do you *read* VGA (and EGA) memory?

Write mode 2 is a bit unusual but not really hard to understand, particularly for those of you who followed my description of set/reset in *PJ*, Volume 5.4. Reading VGA memory, on the other hand, can be stranger than you would ever imagine. Let's start with the easy stuff, write mode 2. If there's room, I'll start in on reading VGA memory, but I suspect that will have to wait until next time.

Write Mode 2

Remember how set/reset worked? Good, because that's pretty much how write mode 2 works. (You *don't* remember? Well, I'll provide a brief refresher, but I suggest that you get *PJ* back issues 5.1 through 5.4 and 6.1 and come up to speed on the EGA and VGA.)

Recall that the set/reset circuitry for each of the four planes affects the byte written by the CPU in one of three ways: by replacing the CPU byte with 0, by replacing it with 0FFh, or by leaving it unchanged. The nature of the transformation for each plane is controlled by two bits. The enable set/reset bit for a given plane selects whether the CPU byte is replaced or not, and the set/reset bit for that plane selects the value with which the CPU byte is replaced if the enable set/reset bit is 1. The net effect of set/reset is to independently force any, none, or all planes to either all ones or all zeros on CPU writes. As we discussed a few articles back, this is a convenient way to force a specific color to appear no matter what color the pixels being overwritten are. Set/reset also allows the CPU to control the contents of some planes while the set/reset circuitry controls the contents of other planes.

Write mode 2 is basically a set/reset-type mode with enable set/reset always on for all planes and the set/reset data coming directly from the byte written by the CPU. Put another way, the lower four bits written by the CPU are written across the four planes, thereby becoming a color value. Put yet another way, bit 0 of the CPU byte is expanded to a byte and sent to the plane 0 ALU (if bit 0 is 0, a 0 byte is the CPU-side input to the plane 0 ALU, while if bit 0 is 1, a 0FFh byte is the CPU-side input); likewise, bit 1 of the CPU byte is expanded to a byte for plane 1, bit 2 is expanded for plane 2, and bit 3 is expanded for plane 3.

Those of you who have been following this series probably understand write mode 2 at this point; I suspect that the rest of you could use some additional explanation of an admittedly non-obvious mode. Let's follow the CPU byte through write mode 2, step by step.

Figure 1 shows the write mode 2 data path. The CPU byte comes into the VGA (or EGA) and is split into four separate bits, one for each plane. Bits 7–4 of the CPU byte vanish into the bit bucket, never to be heard from again. Speculation long held that those 4 unused bits indicated that IBM would someday come out with an 8-plane EGA that supported 256 colors. When IBM did finally come out with a 256-color mode (mode 13h of the VGA), it turned out not to be planar at all, and the upper nibble of the CPU byte remains unused in write mode 2 to this day.

The bit of the CPU byte sent to each plane is expanded to a 0 or 0FFh byte, depending on whether the bit is 0 or 1, respectively. The byte for each plane then becomes the CPU-side input to the respective plane's ALU. From this point on, the write mode 2 data path is identical to the write mode 0 data path. As discussed in earlier articles, the latch byte for each plane is the other ALU input, and the ALU either ANDs, ORs, or XORs the two bytes together or simply passes the CPU-side byte through. The byte generated by each plane's ALU then goes through the bit mask circuitry, which selects on a bit-by-bit basis between the ALU byte and the latch byte.

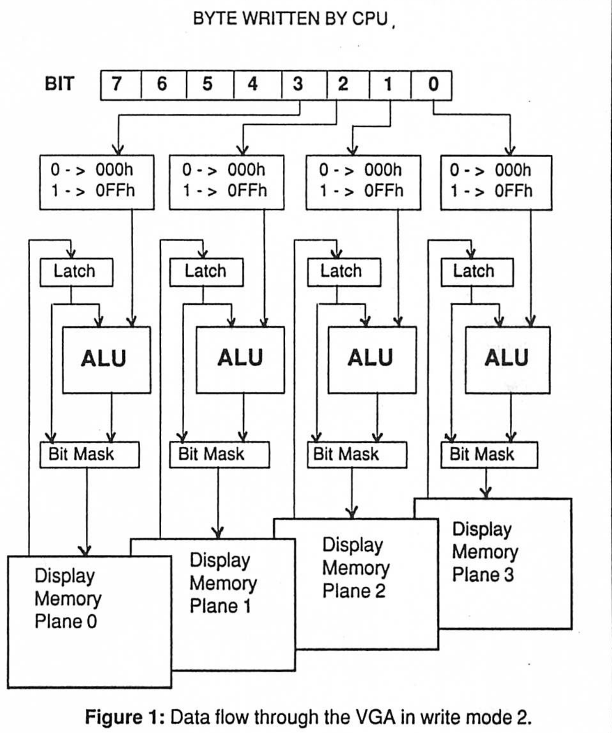


Figure 1: Data flow through the VGA in write mode 2.

Finally, the byte from the bit mask circuitry for each plane is written to that plane if the corresponding bit in the Map Mask register is set to 1.

It's worth noting the two differences between write mode 2 and write mode 0, the standard write mode of the VGA. First, rotation of the CPU data byte does not take place in write mode 2. Second, the Set/Reset and Enable Set/Reset registers have no effect in write mode 2.

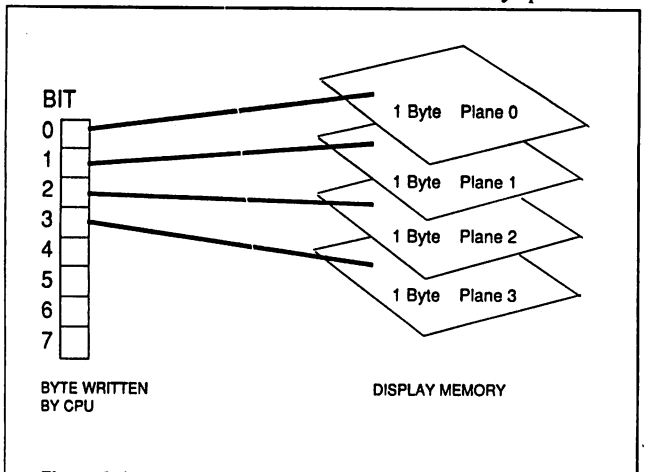
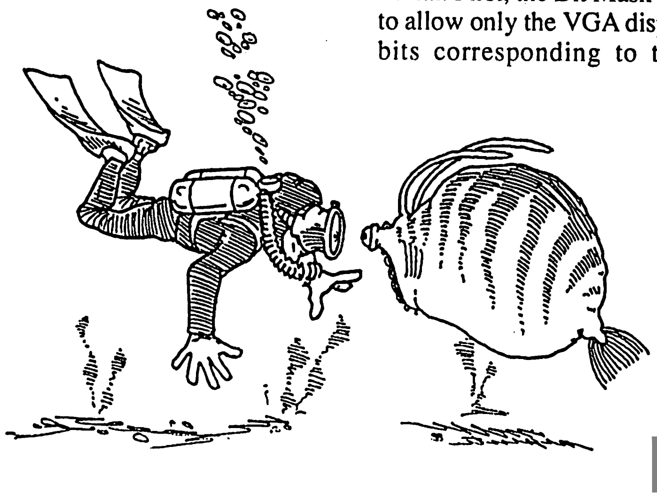


Figure 2: In write mode 2, the lower 4 bits of the byte written by the CPU is written as a color across the four planes, with each bit expanded to a byte.

Now that we understand the mechanics of write mode 2, we can step back and get a feel for what it might be useful for. View bits 3–0 of the CPU byte as a single pixel in one of sixteen colors. Next, imagine that nibble turned sideways and written across the four planes, one bit to a plane. Finally, expand each of the bits to a byte, as shown in Figure 2, so that 8 pixels are drawn in the color selected by bits 3–0 of the CPU byte. Within the constraints of the VGA's data paths, that's exactly what write mode 2 does.

By "the constraints of the VGA's data paths," I mean the ALUs, the bit mask, and the map mask. As Figure 1 indicates, the ALUs can modify the color written by the CPU, the map mask can prevent the CPU from altering selected planes, and the bit mask can prevent the CPU from altering selected bits of the byte written to. (Actually, the bit mask simply substitutes latch bits for ALU bits, but since the latches are normally loaded from the destination display memory byte, the net effect of the bit mask is usually to preserve bits of the destination byte.) These are not really constraints at all, of course, but rather features of the VGA; I simply want to make it clear that the use of write mode 2 to set 8 pixels to a given color is a rather simple special case among the many possible ways in which write mode 2 can be used.

Write mode 2 is selected by setting bits 1 and 0 of the Graphics Mode register (indexed Graphics Controller register 5) to 1 and 0, respectively. Since VGA registers are readable, the correct way to select write mode 2 on the VGA is to read the Graphics Mode register, mask off bits 1 and 0, OR in 00000010b (03h), and write the result back to the Graphics Mode register, thereby leaving the other bits in the register undisturbed. Unfortunately, EGA registers are emphatically *not* readable, making it difficult to take advantage of the VGA's readable registers without separate EGA and VGA drivers.



Copying Chunky Bit-maps to VGA Memory Using Write Mode 2

Let's take a look at two examples of write mode 2 in action. Listing 1 shows a program that uses write mode 2 to copy a graphics image in chunky format to the VGA. In chunky format, adjacent bits in a single byte make up each pixel: Mode 4 of the CGA, EGA, and VGA is a 2 bit per pixel chunky mode, and mode 13h of the VGA is an 8 bit per pixel chunky mode. Chunky format is convenient since all the information about each pixel is contained in a single byte; consequently, chunky format is often used to store bit-maps in system memory.

Unfortunately, VGA memory is organized as a planar rather than chunky bit-map in modes 0Dh through 12h, with the bits that make up each pixel spread across four planes. The conversion from chunky to planar format in write mode 0 is quite a nuisance, requiring a good deal of bit manipulation. In write mode 2, however, the conversion becomes a snap, as shown in Listing 1. Once the VGA is placed in write mode 2, the lower four bits (the lower nibble) of the CPU byte (a single 4-bit chunky pixel) become eight planar pixels, all the same color. As discussed in *PJ*, 5.3, the bit mask makes it possible to narrow the effect of the CPU write down to a single pixel.

Given the above, conversion of a chunky 4-bit per pixel bit-map to the VGA's planar format in write mode 2 is trivial. First, the Bit Mask register is set to allow only the VGA display memory bits corresponding to the leftmost chunky pixel of the two stored in the first chunky bit-map byte to be modified. Next, the destination byte in display memory is read in order to load the latches. Then a byte containing two chunky pixels is read from the chunky bit-map in system memory, and the byte is rotated four bits to the right to get the leftmost chunky pixel in position. This rotated byte is written to the destination byte; since write mode 2 is active, each bit of the chunky pixel goes to its respective plane, and since the Bit Mask register is set up to allow only one bit in each plane to be modified, a single pixel in the color of the chunky pixel is written to VGA memory.

The above process is then repeated for the right-most chunky pixel, if necessary, and repeated again for as many pixels as there are in the image.

"That's an interesting application of write mode 2," you may well say, "but is it really useful?" While the ability to convert chunky bit-maps into VGA bitmaps does have its uses, Listing 1 is primarily intended to illustrate the mechanics of write mode 2. In point of fact, I've heard of a considerably faster (and truly ingenious) way to write chunky pixel bit-maps to EGA memory using write mode 2, read mode 1, and the graphics position registers; sadly, since the VGA lacks graphics position registers, the faster conversion technique works only on the EGA, so it is no longer really useful.

Drawing Color-patterned Lines Using Write Mode 2

A more serviceable use of write mode 2 is shown in Listing 2. The program shown in Listing 2 draws multicolored horizontal, vertical, and diagonal lines, basing the color patterns on passed color tables. Write mode 2 is ideal because in this application color can vary from one pixel to the next, and in write mode 2 all that's required to set pixel color is a change of the lower nibble of the byte written by the CPU. Set/reset could be used to achieve the same result, but an index/data pair of OUTs would be required to set the Set/Reset register to each new color. Similarly, the Map Mask register could be used in write mode 0 to set pixel color, but in this case not only would an index/data pair of OUTs be required, but there would also be no guarantee that data already in display memory wouldn't interfere with the color of the pixel being drawn since the Map Mask register allows only selected planes to be drawn to.

Listing 2 is hardly a comprehensive line drawing program since it draws only a few special line cases, and although it is reasonably fast, it is far from the fastest possible code to handle those cases since it goes through a dot-plot routine and since it draws horizontal lines a pixel rather than a byte at a time. Write mode 2 would, however, serve just as well in a full-blown line drawing routine. For any type of patterned line drawing on the VGA, or indeed for any type of patterned drawing at all, the basic principles remain the same: Use the Bit Mask to select the pixel (or pixels) to be altered, and use the CPU byte in write mode 2 to select the color in which the pixel is to be drawn.

When to Use Write Mode 2 and When to Use Set/reset

As indicated above, write mode 2 and set/reset are functionally interchangeable. Write mode 2 lends itself to more efficient implementations when the drawing color changes frequently, as in Listing 2.

Set/reset tends to be superior when many pixels in succession are drawn in the same color since with set/reset enabled for all planes, the Set/Reset register provides the color data and as a result, the CPU is free to draw whatever byte value it wishes. For example, the CPU can execute an OR instruction to display memory when set/reset is enabled for all planes, thus both loading the latches and writing the color value with a single instruction, secure in the knowledge that the value it writes is ignored in favor of the set/reset color.

Set/reset is also the mode of choice whenever it is necessary to force the value written to some planes to a fixed value while allowing the CPU byte to modify other planes. This is the mode of operation when set/reset is enabled for some but not all planes.

Mode 13h—320 x 200 with 256 Colors

I've been asked several times recently about the programming model for mode 13h, the VGA's 320 x 200 256-color mode. Frankly, there's just not much to it. Mode 13h offers the simplest programming model in the history of PC graphics: a linear bit-map at A000:0000 consisting of 64,000 bytes, each controlling one pixel. The byte at offset 0

Programmers and Information Specialists,

Simplify Your Life.

Learn 3270 HLLAPI

In our intensive ONE DAY SEMINAR you will learn to develop intelligent micro-mainframe links using the PC or PS/2™ and the IBM® industry standard High Level Language Application Program Interface (HLLAPI). Instruction includes both fundamentals and specific programming techniques.

Seminar TOPICS:

- Introduction to Application Program Interface (API)
- Application Ideas and Development
- Learning the Major HLLAPI Services

Participants receive Attachmate's API PROGRAMMER'S TOOLKIT, normally $195, featuring full documentation and programming examples of Attachmate's IBM-compatible HLLAPI.

Seminar fee is $295 per person

Attachmate

Quality Micro-Mainframe Solutions

REGISTER NOW! CALL 1-800-426-6283

Attachmate Corp., 3241 118th S.E. Bellevue, WA 98005, 206-644-4010

IBM is a registered trademark and PS/2 is a trademark of International Business Machine Corporation.

Number 178 on the Reader Service Card

controls the upper left pixel on the screen, the byte at offset 13Fh controls the upper right pixel on the screen, the byte at offset 140h controls the second pixel down at the left of the screen, and the byte at offset 63,999 controls the lower right pixel on the screen. That's all there is to it; it's so simple that I'm not going to waste your time with a demo program. If you do desire a demo program for mode 13h (and modes 11h and 12h as well), a forthcoming issue of the Borland magazine *Turbo Technix* (possibly Volume 1, Number 4, but that's currently up in the air) will provide just that in the form of "The VGA Standard" by Yours Truly.

Flipping Pages from Text to Graphics and Back

In the past week or so I've gotten both a phone call and a letter on an interesting EGA/VGA topic; I didn't remember the name of the caller, but the question is unusual enough that both probably came from the same person. At any rate, the letter came from Phil Coleman, of La Jolla, who writes,

> Suppose I have the EGA in mode $10 (color 640 x 350 graphics). I would like to preserve some or all of the image while I temporarily switch to text mode 3 to give my user a "Help" screen. Naturally, memory is scarce, so I'd rather not make a copy of the video buffer at $A000 to "remember"the image while I digress to the Help text. The EGA BIOS says that the screen memory will not be cleared on a mode set if bit 7 of AL is set. Yet if I try that, it is clear that writing text into the $B800 buffer trashes much more than the 4 Kbytes of a text page; when I switch back to mode $10, "ghosts" appear in the form of bands of colored dots. (When in text mode, I do make a copy of the 4K buffer at $B800 before showing the help; and I restore the 4K before switching back to mode $10.) Is there a way to preserve the graphics image (or at least >24K of it) while I switch to text mode?
>
> A corollary to this question: Where does the 64/128/256K of EGA memory "hide" when the EGA is in text mode? Some, I guess, is used to store character sets, but what happens to the rest? Or rather, how can I protect it?

Those are good questions. Alas, answering them in full would require considerable explanation, and while I intend to do that some day, now is not the time. However, the issue of how to go to text mode and back without losing the graphics image is worth discussing briefly.

Phil is indeed correct in his observation that setting bit 7 of AL instructs the BIOS not to clear display memory on mode sets, and he is also correct in surmising that a font is loaded when going to text mode. The normal mode 10h bit-map occupies the first 28,000 bytes of each of the VGA's four planes. The normal mode 3 character/attribute memory map resides in the first 4000 bytes of planes 0 and 1 (the blue and green planes in mode 10h). The standard font in mode 3 is stored in the first 8K of plane 2 (the red plane in mode 10h). Neither mode 3 nor any other text mode makes use of plane 3; if necessary, plane 3 could be used as scratch memory in text mode.

Consequently, you can certainly get away with saving a total of just under 16 Kbytes—the first 4000 bytes of planes 0 and 1 and the first 8 Kbytes of plane 2—when going from mode 10h to mode 3, to be restored on returning to mode 10h.

That's hardly all there is to the matter of going from text to graphics and back without bit-map corruption, though. One interesting point is that the mode 10h bit-map can be relocated to A000:8000 simply by doing a mode set to mode 10h and setting the start address (programmed at CRT Controller registers 0Ch and 0Dh) to 8000h. You can then access display memory starting at A800:0000 instead of the normal A000:0000, with the resultant display exactly like that of normal mode 10h. (There are BIOS issues since the BIOS doesn't automatically access display memory at the new start address, but if your program does all its drawing directly without the help of the BIOS, that's no problem.)

At any rate, once the mode 10h bit-map is relocated to A800:0000, flipping to text mode and back becomes painless. The memory used by mode 3 doesn't overlap the relocated mode 10h bit-map at all, so all you need do is set bit 7 of AL on mode sets.

Another interesting point about flipping from graphics to text and back is that the standard mode 3 character/attribute map doesn't actually take up every byte of the first 4000 bytes of planes 0 and 1. The standard mode 3 charac-

NEW PRODUCTS! NEW PRODUCTS! NEW PRODUCTS!

ANNOUNCING THE MOST POWERFUL DATABASE GRAPHICS DEVELOPMENT AND ANALYSIS SYSTEM AVAILABLE

G-WIZ®

- **Full featured interactive presentation graphics development system.**
- Works with all popular printers, plotters, cameras and slidemakers.
- Smart scaling routines for ten datatypes provide the fastest route to perfect graphs.
- Fully supports all popular screen devices including: CGA, EGA, VGA, DGIS, and Hercules monochrome.
- Uses maximum output device resolution.
- **Introductory Price, $395.00**

G-RUN™

- **Has all the output features of G-WIZ above plus:**
- Eliminates the need for end-users to learn a graphics package.
- Produces full-featured presentation graphics from within dBASE, R:base, and many other programming languages.
- Thanks to the incredibly robust x-axis and y-axis scaling/labeling routines, this is the ONLY integrated database graphics system available that can produce perfect graphs from live data without operator intervention.
- **Introductory Price, $195.00**

dPROBE®

- **dBASE II, III, and III+, Foxbase, and Clipper source code documentation and analysis.**
- More than 13 different reports on program/variable structure available.
- Time-saving table of contents organized in two different ways!
- An invaluable aid to consultants and application developers! Completely automates program documentation and structure.
- **Introductory Price, $89.00**

Offered to you by MIGHTYSOFT™Inc.,P.O. Box 51048, Seattle, Washington 98115. To order call: (206) 525-3766. Telex: 6503323229. Visa and Mastercharge welcome.

Number 179 on the Reader Service Card

ter/attribute map actually takes up only, every even byte of the first 4000 in each plane; the odd bytes are left untouched. This means that only about 12 Kbytes actually have to be saved when going to text mode. The code in Listing 3 flips from graphics mode to text mode and back, saving only those 12 Kbytes that actually have to be saved. This code saves and restores the first 8K of plane 2 (the font area) while in graphics mode, but saves and restores the 4000 bytes used for the character/attribute map in text mode since the characters and attributes, which are actually every other byte of planes 0 and 1, respectively, appear to be contiguous bytes in memory in text mode and so are easily saved as a single block.

Explaining why only every other byte of planes 0 and 1 is used in text mode and why characters and attributes appear to be contiguous bytes when they are actually in different planes is a large part of the explanation I haven't room to go into now. One bit of fallout from this, however, is that if you flip to text mode and preserve the graphics bit-map using the mechanism illustrated in Listing 3, you shouldn't write to any text page other than page 0 (that is, don't write to any offset in display memory above 3999 in text mode) or alter the Page Select bit in the Miscellaneous Output register (3C2h) while in text mode. In order to allow unfettered access to text pages, it would be necessary to save the first 32K of each of planes 0 and 1. (On the other hand, this would allow up to 16 text screens to be stored simultaneously, with any one displayable instantly—a topic for yet another installment.) Moreover, if any fonts other than the default font are loaded, the portion of plane 2 those particular fonts are loaded into would have to be saved, up to a maximum of all 64K of plane 2. In the worst case, a full 128K would have to be saved in order to preserve all the memory potentially used by text mode.

As I said, Phil Coleman's question is a complex one, and I've only touched on the intriguing possibilities arising from the various configurations of display memory in VGA graphics and text modes. Someday I'll return to it, but for now we've still got the basics of the remarkably complex VGA to cover.

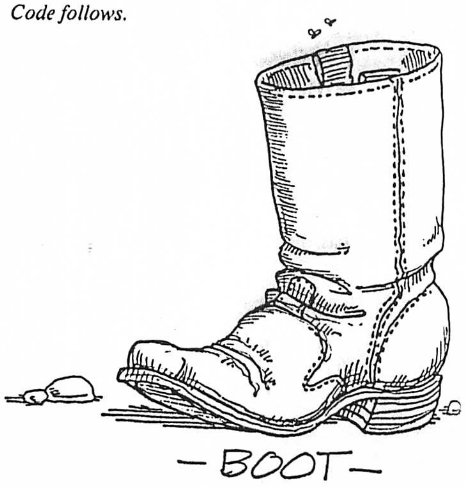
Until Next Time

As I suspected, we didn't get around to learning how to read VGA memory. That's probably just as well since that leaves the whole next installment in which to cover that fairly esoteric subject. If there are any other EGA or VGA topics you'd like to see covered, write to me in care of *PJ* or at the post office box listed below, or drop me a line on MCIMail (user name: MABRASH), and I'll try to get to it soon. See you next time. ♦

Michael Abrash is an Engineering Fellow currently working on advanced graphics projects for Video Seven, Inc., in Fremont, California. Readers may write to Michael at

P.O. Box 390351
Mountain View, CA 94039

Code follows.



If you see something here you like, by all means, compare it with what our competition has to offer.

Seidl Version Manager (SVM) Version 2.0 is a comprehensive source code configuration system designed for professional software developers. SVM tracks changes made to individual modules (revisions) and stores "version" information that tells which revisions go into making which versions of a product. SVM supports multiple developers in both networking and non-networking environments. Some of its features include: revision branching, revision merging, user ids and optional passwords, seven different file access levels, a menu driven shell, formatted reports with user definable layouts, central journal logging, on-line help, optional data compression, a remote site update utility, and more. ***$299.95, LAN version $1000.00***

Seidl Make Utility (SMK) Version 2.1 is the most sophisticated product generation utility available. SMK is not just another copy of the of the UNIX make. SMK uses a structured, high-level, *dependency definition language (DDL)* that is a programming language in itself. It supports symbolic constants, parameterized macros, for loops, if-then-else style conditionals, interactive statements, ambiguous path name expressions, and much more. In addition, SMK is lightening fast and won't bog down, even on your largest projects. ***$99.95, LAN version $500.00 (includes SMKgen).***

New!* SMKgen Version 1.0** is an automatic dependency generation utility. By examining the files that make up a product, SMKgen determines how those files depend on each other and amazingly writes a dependency file that can be processed by SMK. SMKgen can also construct link files and *filelist* files that integrate with SVM. With SMKgen, the task of placing a software project under SMK is reduced to executing a single command. ***$49.95

"SVM is a full-featured system that has all the essential capabilities you are likely to need, and is further distinguished by several important features that none of its competitors yet have." ***- The C Users Journal™, February, 1988.***

When you know the facts, you'll know we're the best.

For more information call: **1-313-662-8086**
For a free copy of our competitive comparison report, write to us on your company letterhead. MC/VISA/COD accepted.

SEIDL COMPUTER ENGINEERING

3106 Hilltop Dr., Ann Arbor, MI 48103

Number **174** on the Reader Service Card

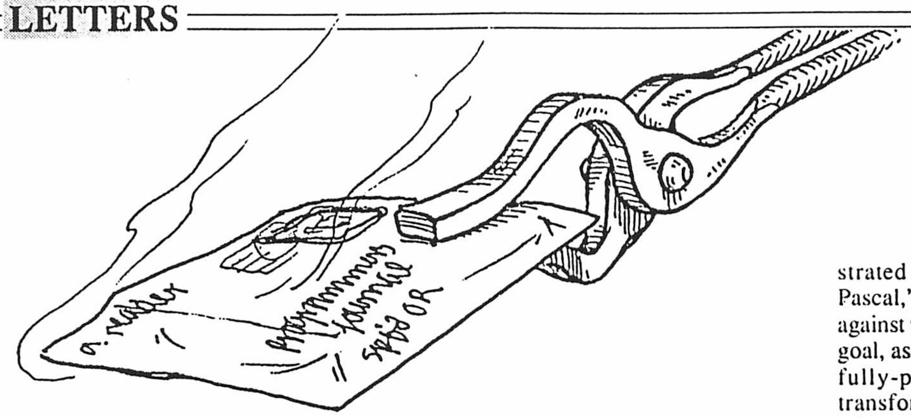


Diagram for the EGA

After reading Michael Abrash's articles on the EGA (*PJ*, Volumes 5.1–5.4) and carefully rereading the *IBM Tech Ref* for the EGA Adapter, I came up with the enclosed diagram. It shows functionally what happens when you read or write to the EGA display in graphics mode for plane 0—the other planes are identical. This helps me immensely when I'm programming at the hardware level.

Please review it. You are free to publish it if you think it would help your readers.

Bob Montgomery
Orlando, FL

Michael Abrash Replies

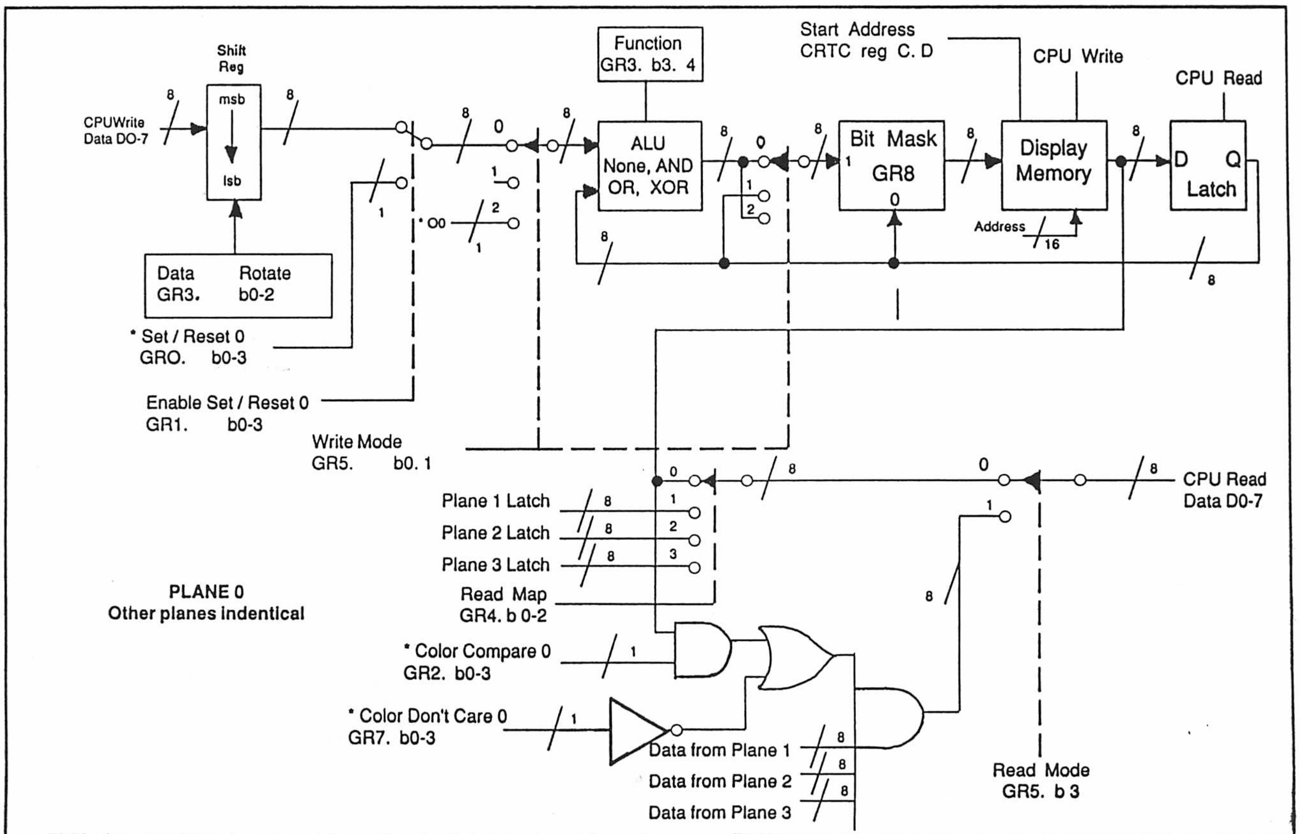
I'm no hardware engineer, but Bob Montgomery's diagram looks great to me. It should help a lot of people understand the EGA.

Michael Abrash
Mountain View, CA

Jon Greenblatt Goes Too Far (Even for Jon Greenblatt)

I'd like to respond to some comments in the Letters column in Volume 5.6. I would argue that "ludicrous" and "appalling" optimizations (readers' descriptions of those I demonstrated in "A Note on Optimizing Turbo Pascal," Volume 5.5) have to be measured against what you're trying to achieve. If your goal, as was ours at ELFSOFT, is to write a fully-programmable, self-customizing, transformational natural language parser, with a back-end that generates efficient dBASE code, there are some serious constraints. To be co-resident with dBASE (on 512K systems) required limiting ourselves to 250K. Furthermore, according to Winograd, Schank, Sager, *et. al.*, this program is impossible on its face—a futuristic dream. We weren't content merely to demonstrate it theoretically, but wanted to make it practical, which meant turbo-charging it until a complex query that generates two screens of .PRG code would translate in (average) 15 seconds.

This was necessary because however ludicrous or appalling it is, a manager who asks an employee to find out "how many of the *Programmer's Journal* prospects come



Bit Mask passes latch data where bits are 0 and switch data where bits are 1.
GR => Graphics Register (3CEh - 3CFh).
* => extend bit n for plane n to a byte.
CPU write data is rotated before any other operation is performed.
Color Don't Care bit n = 0 makes color compare ignore plane n.
Read Map register = n selects plane n for reading.

Bob Montgomery			
Title: EGA Read / Write Functional Diagram			
Size: A	Document Number		REV
Date: November 20, 1987		Sheet	of