Introduction to VGA Mode 'X'

Robert Jambor

 DOWNLOAD ... *The example file mentioned in this article is contained in the file **MX_SCROL.EXE** (3,799 bytes) which can be downloaded by clicking this disk icon.*

Preliminary notes

This article contains a fair amount of technical information and does not attempt to describe what registers are, how addresses on an Intel 80x86 processor work, or anything like that. I have enough material here to warrant leaving that sort of detail out. Phil has covered many of those topics in previous GDM articles, so if you have trouble, you could refer to those instead.

Mode 13h revisited

Before getting into the nitty-gritty of sprite engines etc, we must understand what Mode X actually is.

In the previous GDM articles, Phil discussed Mode 13 to the readers very nicely. One of the points he made was that in Mode 13, the addressing of pixels was in a linear fasion. That is, increasing our address in video memory by 1 byte resulted in the address of the next pixel. This can be really handy for writing very optimised assembly code using the assembly language string instructions.

However, Mode 13 has a few quirks which tend to make it unusable for serious game work. The biggest problem is that with Mode 13, you can only have one video page. This is a big problem since standard VGA cards today have at least 256k of video memory (some early cards had only 64k!). Looking at Mode 13 we see that it uses up 320x200 pixels = 64000 bytes, or approximately 64k. We appear to have lost the remaining 192k of video memory that we paid for. Using Mode 13 we cannot even access this memory let alone use it as a second, third or fourth video page.

Mode-X lives!

To overcome the problems with Mode 13, we use Mode-X (or tweaked VGA as it is also known), that is we re-program the VGA registers to basically do our bidding. Most inexperienced programmers would balk at the mere mention of programming these registers but I will show you that it is not as hard as it sounds. If you want to start from scratch, you will need a good book covering the aspects of the VGA registers. One such excellant book is the "Programmers Guide to the EGA and VGA cards" by Richard Ferraro.

"If you want so start from scratch, you will need a good book"

Firstly, if we are going to program the VGA registers then we should know what we are going to program them with. The VGA registers (of which there are more than 50) contain information regarding the state of the display. The registers fall into 5 basic groups :-

1. General or external registers.
2. Sequencer registers.
3. CRTC (Cathode Ray Tube Controller) registers.
4. Graphics registers.
5. Attribute registers.

To setup Mode X, we will mainly need to make changes to the CRTC registers, but we will also need to make a few minor changes to some of the other registers too.

Before describing how this is done, I should make mention that we do not reprogram the VGA from scratch since doing so is tedious at best and inefficient anyway since we have the BIOS there to do all the really boring stuff.

So, to switch into Mode X, we simply call the BIOS's set video mode interrupt to switch us into Mode 13. Once we are in Mode 13, we make adjustments to many of the CRTC registers to switch us into Mode X.

On the VGA there are 26 CRTC registers in all. Instead of making changes to the registers here and there, we create a table of values containing the information needed for each CRTC register we intend to change and copy the entire table to the VGA. One thing that should be realised when doing this, is that our VGA card must be 100% register level compatible with the VGA standard. This should not be a problem anymore but some earlier cards were only BIOS level compatible with VGA.

"Mode X ... corresponds to a group of resolutions"

A further note about Mode X is that is does not correspond to a particular resolution per se, but corresponds to a group of resolutions capable by the VGA hardware. I chose the 360x240 resolution mode because of it's higher resolution over 320x200. As such, the following values are based on that selection.

Before proceeding to "quote" the table of values, I will briefly list each register's function so that you can see what each value does. I originally started describing each register but that was taking up way too much space and probably would not have served much use anyway.

Register	Bits	Name	360x240 Value
0	0-7	Horizontal Total	107
1	0-7	Horizontal Display End	89
2	0-7	Start Horizontal Blanking	90
3	0-4	End Horizontal	14
	5-6	Display Enable Skew	0
	7	Compatible Read	1
4	0-7	Start Horizontal Retrace	94
5	0-4	End Horizontal Retrace	10
	5-6	Horizontal Retrace Delay	0
	7	End Horizontal Blanking, 6th bit	1
6	0-7	Vertical Total Register	13
7	0-7	=Overflow register=	-
	0	Vertical Total 8th bit	0
	1	Vertical Display End 8th bit	1
	2	Vertical Retrace Start 8th bit	1
	3	Start Vertical Blanking 8th bit	1
	4	Line Compare 8th bit	1
	5	Vertical Total 9th bit	1
	6	Vertical Display Enable 9th bit	0
	7	Vertical Retrace Start 9th bit	0
8	0-4	Preset Row Scan	unused
	5-6	Byte Panning	unused
9	0-4	Maximum Scan Line	9
	5	Start Vertical Blanking 9th bit	1
	6	Line Compare 9th bit	0
	7	200-To-400-Line Conversion	0
A	0-4	Cursor Start	unused
	5	Cursor On/Off	unused
	6-7	*Not Used (keep as 0)*	-
B	0-4	Cursor End	unused
	5-6	Cursor Skew	unused
	7	*Not Used (keep as 0)*	-
C	0-7	Start Address High	unused
D	0-7	Start Adress Low	unused
E	0-7	Cursor Location High	unused
F	0-7	Cursor Location Low	unused
10	0-7	Vertical Retrace Start	234
11	0-3	Vertical Retrace End	12
	4	Clear Vertical Interrupts	0
	5	Disable Vertical Interrupts	1
	6	Bandwidth	0
	7	Protect Registers 0-7	1
12	0-7	Vertical Display End	223
13	0-7	Offset Register	45
14	0-4	Underline Location	0
	5	Count By Four	0
	6	Double Word Mode	0
	7	*Not Used (keep as 0)*	-
15	0-7	Start Vertical Blank	231
16	0-6	End Vertical Blank	6
	7	*Not Used (keep as 0)*	-
17	0-7	=Mode control register=	-
	0	Compatability Mode Support	1
	1	Select Row Scan Counter	1
	2	Horizontal Retrace Select	0
	3	Count By Two	0
	4	Output Control	0
	5	Address Wrap	1
	6	Word/Byte Mode	1
	7	Hardware Reset	1
18	0-7	Line Compare	unused

Whew!, that was fun but I wouldn't like to do it again. That's the values we will need to program the VGA into Mode X using a resolution of 360x240. I have listed the names of some registers which aren't really needed to setup the mode but are documented for completeness sake.

"Apparently, it IS possible to cause damage to monitors if incorrect values are used!"

A word of warning about programming these registers directly I should mention is that there are some values which are not allowed for some registers and also there are some registers which cannot be used in conjunction with others, so care must be taken when programming them. Apparently, it *IS* possible to cause damage to monitors if incorrect values are used, however I suspect that a monitor would probably have to be subjected to these "out-of-sync" signals for a reasonable amount of time to do any real damage.

On a less technical note, I would like to relate an incident that I came across on the PASCAL echo on Fidonet:

An inexperienced programmer had somehow figured out how to use the OUT instruction to set up some values on his VGA card. He soon released a public domain "screen saver" program which would blank the screen for you. What he didn't tell people initially was that to arrive at this program, he sent junk data to some of the CRTC registers. He noted that a certain combination of values could be used to blank the screen. He didn't realise that the screen went blank because the poor monitor was thrown out of synchronisation by the data sent from the VGA card.

"A little knowledge can be dangerous, so be careful!"

This is one example of how a little knowledge can be dangerous, so be careful with your Mode X init procedures. :-)

The real thing

We have now done our research and have the values for our Mode X initialization, where do we go from here? Firstly, the code I will be presenting is in assembler primarily because this particular function is best suited to that language. In assembly language there exists a command called OUT which sends either a byte or word (or DWORD on 386 & higher processors) to an output port.

The syntax for its most common use is as follows:-

```
        OUT DX, AL
    or  OUT DX, AX
```

where DX is the port which will receive the data and AL/AX is the byte/word of data to send. An interesting side effect with the second mode is that the low byte (AL) is sent to the port specified by DX, however the high byte (AH) isn't. It is instead sent to the port specified by DX+1. This may seem like a dumb "feature" but it does have it's uses, as I'll now explain.

The VGA has over 50 registers but not all of them have their own unique port address. Many of them are multiplexed onto one "shared" port address to save on hardware. When registers are multiplexed in this way, there are usually two registers which have unique port addresses and allow access to the "real" registers. The first of these registers is called the "Address" register and the second is called the "Data" register. To set the value for one of our "real" registers, we first tell the address register which one of our registers we would like to set up and then send the data for it through the data register.

Example

The colour VGA CRTC registers have an address register at port address 03D4h and a data register at 03D5h, ie. at the next port address. Suppose we want to send the first byte of info for our Mode X init routine, ie. the "Horizontal Total" value of 107 to the appropriate register, ie. register 0. One possible way to do this is as follows :-

(Once again, in assembler)

```
        MOV DX, 03D4h   ; Port address of CRTC Address register.
        MOV AL, 00h     ; Register number 0 to change.
        OUT DX, AL      ; Inform VGA.
        INC DX          ; Point DX to the CRTC Data register.
                        ; (remembering that it is the next one)
        MOV AL, 06Bh    ; Value for Horizontal Total. (6Bh=107d)
        OUT DX, AL      ; Inform VGA.
```

This will work ok, except that it is very inefficient, even though we used INC DX instead of reloading DX directly with 03D5h. A better way to accomplish the same goal is as follows :-

```
        MOV DX, 03D4h   ; Port address of CRTC Address register.
        MOV AX, 06B00h  ; Put register number (00) in AL &
                        ; register data (6B) in AH.
        OUT DX, AX      ; Output combined data to VGA.
```

This has the effect of sending the value 00h to port 03D4h, ie. the CRTC address register, and the value 6Bh to port 03D5h, ie. the CRTC data (Horizontal Total) register. This is much more efficient and also has an added bonus when working within a loop. With the original code we had to increase DX by one with the INC DX line, but if we want to send more data, (as in a loop) we would have to restore DX to 03D4 by using DEC DX. If we use the second example then we can leave out both the INC DX & DEC DX lines altogether.

All that is left now is to tabulate our values along with their register numbers and then code a loop to send each word out to the VGA in sequence. I won't go into much detail here on how to do this, as I have included a source code file called MX_INIT.ASM to show you how this is done. Note that I have *NOT* tried assembling this particular file and is is presented purely for reference. It should work however, since 90% of it came directly from my sprite engine software.

Before leaving this topic, there are a few more registers which need to be set up prior to our modifying the CRTC registers, as noted in the source code. They are as follows:

1. Sequencer Register 4 should have a value of 6. This effectively disables the Chain by 4 mode used to give Mode 13 it's "linear" addressing.

2. The sequencer must then be reset via an asynchronous reset. I am not 100% certain why but I think it is so that the VGA applies the change made in 1 above.

3. We must set up the clocking frequencies for our screen by fiddling with the Miscellaneous Output register.

4. As you may have noticed in the table of registers above, their is a bit called the "Protect Registers 0-7" and as it's name suggests, it write protects CRTC registers 0-7. We must first disable this before sending our data and then re-enable after we are finished.

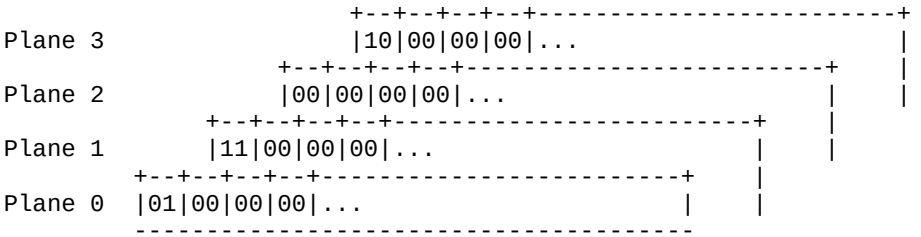
"After reading ... you may wish you had never heard of mode X"

Well that's about all there is to initializing a Mode X screen using assembly language. The next section will describe the differences between Mode 13 and Mode X and after reading it, you may wish you had never heard of Mode X...

Mode X vs. mode 13h

So you thought you had graphics mastered when you wrote that filled circle routine a couple of days ago, well then, welcome to Mode X. As Phil had described in GDM1 (I think), the Mode 13 screen is basically arranged as a table of colour values, the width of the table being 320 and the height 200. This was a very nice way of arranging the screen since to change the colour of a pixel, all that needed to be done was change one of the values in the table. Unfortunately however, in Mode X things are not quite so easy.

The Mode X screen is arranged as four tables (planes), each of which is made up of 90 columns and 240 rows (for our 360x240 pixel resolution). The colour information for each pixel spans across all four planes of video memory. Each colour value is eight bits since we are allowed 256 colours and as a result, each video plane contains two bits of colour information for each pixel. To illustrate this properly, we definately need a diagram :-



The above diagram is not very good, but it should convey the general idea behind what is termed PLANAR memory addressing. This is the method used by Mode X and also by EGA/VGA 16 colour modes, so theory on this type of addressing should be fairly abundant. In the diagram, the pixel at coordinates (0,0) has a colour value of 10001101 in binary or 141 in decimal. Notice how this value is spread across the four video planes; bits 0&1 are on plane 0, bits 2&3 are on plane 1, 4&5 on plane 2, etc. The next pixel (at coordinates (1,0)) has bits 0&1 back on plane 0 and are stored directly after the first pixel's bits 0&1.

Now, since each byte of display memory is eight bits wide, and since each pixel has 2 bits of information on every display plane, that means that each byte of display memory indirectly manipulates 4 pixels. Some programmers were raving about Mode X's 4x speed advantage over Mode 13 because of this ability, but this increase can only be realised in certain operations.

One further point of interest is that each of the video planes occupies identical address spaces in the system address space. This means that, to manipulate the video data, we must inform the VGA which plane we wish to modify so that it can "bank in" that plane for use. Once the plane has been banked in, we can manipulate the information on the plane through normal memory addressing procedures. The important thing to note here is that only one plane can be modified at a time. This point should not be taken as FACT since it is possible to modify several pages at once to perform some advanced operations, but for now I would like to keep things simple.

Extra goodies with mode X

Although the Mode X memory layout may seem confusing, there are a number of advantages with working in this mode. Firstly, since the display memory is PLANAR (ie. spread across planes), only one plane needs to be banked in at a time. Each plane only contains a quarter of the display information, so for a 320x200 pixel resolution screen which we know takes up 64000 bytes, each plane only contains 16000 bytes of that information. So what happens to the other 48000 bytes (per page) of unused memory?

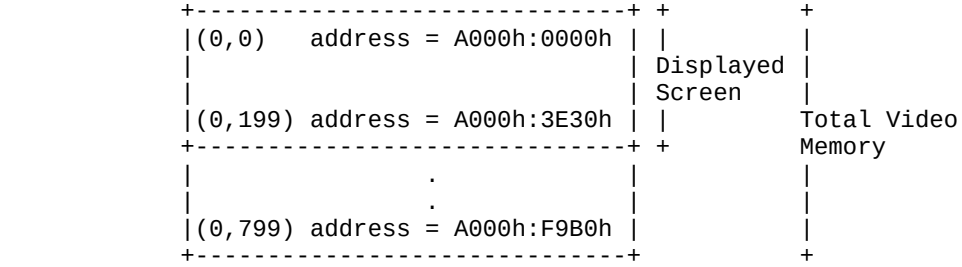
Well I'm glad you asked, remember that 192k of missing display memory we discussed earlier, well here it is. A second important feature when working in this mode is a speed increase when doing certain operations. The VGA has a 32 bit internal latch which it uses to transmit data to/from the VGA/CPU. The latch is most useful when we are transferring data from one portion of video memory to another, since the data is transferred 32 bits at a time, and not 8 or 16 bits at a time, as is the case when transferring data to/from the CPU from/to the VGA. This makes it easy (and fast) to copy data from one area of video memory to another and in particular makes it easy to write "page-flipping" animation, but more on that another time.

Pages, pages everywhere

As I mentioned above, we now have some more video memory to play with and there are many ways to do just that.

Before explaining the page addressing/scrolling features, it would be handy to imagine the planes of video memory as four playing cards, one behind the other, thus occupying the same memory addresses (or space), though not at the same time. To access the data contained within any plane, you would first need to bring it to the front, since you couldn't normally see what was on it.

Keeping this in mind, recall that our planes only take up 16000 bytes each to make up a 320x200 pixel screen. The rest of the data (ie the other 48000 bytes) is actually off-screen and can either be paged or scrolled in using the same technique. Try to think of our 320x200 screen as a window on a much larger screen, ie a 320x800 pixel screen which is four times as large, taking up the whole 64000 bytes per page. The actual position that our "window" can be in is easily configurable through the VGA's SAL (Start Address Low) and SAH (Start Address High) registers.



The above diagram hopefully explains what I am on about here. The "Displayed Screen" rectangle is the area of memory that is currently displayed on the screen and is 200 lines high. However we can shift where the top left corner of this window is actually positioned within video memory by changing the SAL & SAH registers.

"Making changes to these registers can yield scrolling and paging"

Making changes to these registers can yield two things, scrolling and paging. If we set the start address to 0, our window will be at the top of our display memory. Increasing the value in SAL/SAH by 80 moves our window down by one line. This is because there are 80 bytes of pixel information per line, since each pixel uses only 2 bits per plane and 320 x 2 bits = 80 bytes. Note that our discussion here is for a 320x200 mode screen, not a 360x240 mode screen as we described previously. A 360x240 pixel screen has 90 bytes per line since 360 x 2 bits = 90 bytes.

To implement paging we simply divide up our video memory into four sections, each 200 lines high and set the start address to the top of each page. For example, page 0 would have a start address of 0000h, page 1 would start at line 200 and would have a start address of 3E80h or 16000 (80 bytes per line * 200 lines) The next two pages would have starting addresses of 7D00h (32000) and BB80h (48000) respectively.

Note that we only tell the VGA controller where the offset is in video memory, not the whole address. The VGA graphics screen still begins at segment A000h, however we are telling it to change where it starts reading in video data for display purposes. In our paging example above, our page 0 data is located at the absolute address of A000h:0000h, page 1 at A000h:03E80h, etc.

The VGA's point of view

As explained above, the VGA's starting address offset is controlled by the SAL & SAH registers, which if you examine our table earlier can be seen to be register C & D. SAL & SAH actually combine to form one 16 bit register (since SAL & SAH are both eight bits wide). This 16 bit register holds the value of our offset as described above, however we will still need to send out the low and high 8 bits to the VGA separately.

Consider what would happen if there was a delay between sending the low and high bytes to the VGA. During this delay period, the VGA may have an incorrect starting address value, since the high value is yet to be set up. Since the combination of SAL & SAH may point to an area of the screen which is somewhere off screen, during the delay the VGA will be displaying video information from the wrong source addresses. Once we send out the high byte, the VGA will be fine again and will be displaying data from the correct source and our screen will be as we expected it. You may think that this is ok since, "in the long run" we got what we wanted, but that unwanted data that was appearing on our screen during the delay period will appear as an unwanted flicker which is bad news for any animation.

To get around this problem, we wait until the screen is in it's vertical retrace period. This is the time when the monitor's electron beam (the one that lights up all those lovely pixels) moves from the bottom right hand corner of the screen back up to the top left hand corner. During this time, the electron gun is switched off, so that no dots on the screen are changed. This event happens about 60-70 times a second and this refresh rate is usually referred to in Hertz or hz for short. This means that a complete screen update will take from 1/70 - 1/60 of a second, quite fast indeed. Any routine that waits for the refresh period to do it's work must get its job done quickly, taking only as long as is required.

One further precaution that should be taken is the disabling of interrupts during our change of SAL & SAH. This is to ensure that we are not interrupted during our changing of these registers, since there is no guarantee that the interrupt routine will be finished it's processing before the VGA finishes it's retrace period. As a point of interest, this actual retrace period is called the Vertical retrace period since there also exists a Horizontal retrace period. This occurs after the electron beam has completed highlighting a row of pixels (including the border) and needs to move to the start of the next line.

Both the above ideas apply to many areas in VGA programming, most notably in palette cycling which is a fairly time consuming process since we need to shift 768 bytes from the system ram to the VGA registers during the refresh period. Although 768 bytes may not sound like much, copying them to the VGA card through the ISA bus controller chip is SLOW since the bus only runs at a fixed 8Mhz and is only 8 or 16 bits wide. Local Bus has helped a lot in this area, but not everyone has Local Bus yet. :-(

"Michael Abrash's column introduced most of us to Mode X in the first place"

Now that we have covered the basics of this particular VGA function, we should do something to prove that we have understood at least some of it. So, to do just that I have included a small program written in Turbo Pascal and Inline Assembler to initialize the Mode X screen, fill it with random data and scroll it up and down some. The program is called MX_SCROL.PAS, and don't worry if you can't understand it all because there is still a lot to Mode X that we have to learn. As you can see from the size of the source code, it doesn't take a lot of power to do this particular task & the resulting program shows a very smooth scrolling background. This scrolling effect is a bit of a "parlour trick" though, since it isn't very useful in itself. This is because scrolling the whole screen means scrolling any status lines, score displays, sprites, etc. which is undesirable. Don't be put off however, the effect can and is used in some games, however a lot more code is needed before we can write any "gameable" code using this technique.

(The demo program is in the MX_SCROL.EXE file. I seem to have lost the source - Robert, contact me if you can! Phil).

Finale

Well, that's about all I have for this article, but stay tuned because I would like to contribute more articles to GDM on sprites and further articles on Mode X, if space permits in this excellant electronic magazine. Sorry if I stuffed up any values in the register table but I was disecting my Mode X init routine to extract the various bits. Check the source code if there is a problem.

References

"Programmer's Guide To The EGA And VGA Cards"
2nd Edition
Richard F. Ferraro
ISBN 0-201-57025-4

Credit where it is due

I did not come up with the values for the VGA CRTC registers, but I did make a few alterations to them. I owe credit to Michael Abrash since his column in Dr. Dobbs introduced most of us to Mode X in the first place. Also, someone (lost the original message) in the Z3_PASCAL echo had given me some help in initializing Mode X with a screen width of 360 pixels. I simply combined this with Michael's code which initializes the screen to 240 lines (at 320 pixels wide) to arrive at my combined 360x240 mode.

About the author

I was introduced to computers through an adventure game called "House" on an old TRS80 microcomputer. A friend and I used to play it every day after school for some time. I then bought an Amstrad CPC664 computer through which I tought myself BASIC and some assembly language using the Z80 chip. In high school I got my first real taste of PC's, using Sperry XT's with no graphics capabilites. The computers were still excellant tools with which to build up a base of knowledge on. After high school, I went to College/University and emerged with a degree in Applied Science, majoring in Mathematics and Computing (concentrating more on computing). From there I began working at a BHP-Utah Coal Ltd (Now known as BHP Australia Coal Pty Ltd) laboratory, writing programs for capturing data from various instruments using serial ports, A/D cards, etc. During my uni years, I had also worked at Sugar Research Institute in Mackay, working on an animated cane railway scheduling simulator using Sun Sparcstations. In my spare time, I research and tinker with graphics and sound on the PC, trying to teach myself more and more about the VGA hardware. There is just so much to learn in this field that specialising is definately the key. I someday hope to release some sort of decent VGA game programming toolkit to the public (ie. Free, none of this "shareware" garbage) via BBSs, however I think I am a little way off that goal yet. My current personal project is a sprite graphics engine using Mode X at a resolution of 360x240 pixels, and I have made a fair bit of progress on this.