XBin tutorial

Introduction.

XBin offers many features several coders have never programmed before. For this reason, I'll cover these aspects of text mode programming within this small tutorial.

A rough understanding of assembly is required. Source code is provided in assembly. The reason for the choice is obvious, most of the code involves dealing with VGA registers and bit twiddling, two things assembly is the best choice for.

I'll be covering most of the features at a novice to moderate programming level. In case you don't understand some parts, it may be a good idea to grab a couple of books on VGA programming. This text is by no means the ultimate reference for the VGA. Some exceptions excluded, I'l usually be describing **HOW** the overall concept works, and not enter into details on how the VGA hardware actually works, you'll need to consult a decent reference for that.

NOTE ! All the assembly sequences assume that any register may be freely modified Depending on language used this is not usually the case (Pascal requires you to restore SS, SP, BP & DS to their original values upon leaving the Assembly code. Most C compilers require that on top of those four registers, you also keep SI and DI intact). Check the manual of your compiler for specifics. Most compilers allow for inline assembly code, this way it should be fairly easy to use the features described here

The code samples are for VGA **ONLY (!)** do NOT use them on an EGA, most codesamples won't work on EGA. The code only modifies what it needs to perform it's job, when only one bit needs to be modified, the others are preserved.

Please note: I did not have a chance to test all of the code on it's workings, most of it is taken directly out of the XBin viewer (See SimpleXB.Pas in the XBin Developers Kit), so should be functional, if you use the source listed HERE however, some minor mistakes may have slipped in. The XBin Viewer should be a fairly good reference on how to implement a basic XBin viewer.

VGA in general.

The VGA usually behaves as a pretty simple device. Textmode and 256 color graphics modes are straightforward and easy to work with. The VGA BIOS usually offers enough functionality to make use of most of what the VGA has to offer.

Unfortunately the VGA BIOS is also notoriously slow. Especially when writing a protected mode program, using the BIOS often to perform simple operations can literrally slow your program to a crawl.

In such cases, or when the BIOS simple doesn't cut the cake, you will have to program the VGA directly. This is usually where the mess starts.

While this small tutorial is far from a complete reference for the VGA, it should be more than adequate

when you want to make full use of the XBin format. All the sample code is functional. If you REALLY want to know what exactly is going on, then you should consult a VGA reference.

Some good books I have read myself and recommend to anyone involved in graphics and/or demo programming are:

- Programmer's Guide to the EGA & VGA Cards by R. Ferraro (Addison-Wesley)
  This book is one of the most complete VGA references available. It is currently in it's third print, and involves everything from EGA, to VGA, to SVGA and even covers the most commonly used SVGA accelerator chips. Even though it lacks sample code to explain most of what it covers, it remains something to have lying beside you when coding graphics.
- Zen of graphics programming by Michael Abrash (Coriolis group books)
  A very easy to read book covering most of the aspects of the VGA hardware, textmode, 16 color graphics, 256 color, and "tweaked mode". You'll also find some pretty fast code for lines, polygons, circles, sprites and so on. It even has a complete 3D library included. It's a book I strongly recommend to anyone who needs a little push in the back to get going with graphics. The sample code is usually listed in a simple C version, and then goes to several fases of optimisation (usually ending up with high-performance assembly code).
- PC & PS/2 video systems by Richard Wilton (Microsoft Press)
  UGH! Am I actually advertising something by Microsoft... Ah well... Like Zen of Graphics programming, this book also covers most of the aspects of the video hardware. This book does however also covers MGA, CGA, Hercules, Hercules plus, EGA, MCGA and VGA. Because everything but the VGA have become near extinct, you may end up using only a small portion of the book. The sample code (Basic, Pascal, C & ASM) however is usually of pretty good quality.

Setting the VGA palette in text mode.

This is one of the features most of the programmers know pretty well, and indeed, in 256 color graphics, where this feature is used most, it's pretty simple to do. Each pixel color nicely translates to a palette Red, Green, Blue value.

In textmode, things are slightly more complicated. In order to be compatible with the EGA, in textmode the 16color EGA palette is used. The EGA palette then maps into the 256 color VGA palette to find the RGB values to send to the monitor.

Unfortunately, the 16 color EGA palette does NOT map to the first 16 colors of the VGA palette. Fortunately though, we CAN program the EGA palette to do just that.

Setting the 16 color RGB palette stored in an XBin takes two steps.

1. Set the EGA palette to map to the first 16 colors of the VGA palette.
2. Set the first 16 RGB values in the VGA palette.

Sample Code to set the palette located in XBIN.PAL:

```
   MOV   DX,003DAh   ; CRTC Status register
   IN    AL,DX       ; Read CRTC Status. (This will reset
                     ; the Attribute controller flip-flop)
; ===== Set EGA palette to color 0-15 =====
   MOV   DX,003C0h   ; Attribute controller
   MOV   AL,0        ; Color 0. (and Clears PAS field)
NxtPalEGA:
   OUT   DX,AL       ; Set paletteregister to change
```

```
   OUT   DX,AL       ; Set value for palette register
   INC   AL          ; Next color
   CMP   AL,0Fh      ; All colors done ?
   JBE   NxtPalEGA   ; Nope.
   MOV   AL,20h
   OUT   DX,AL       ; Set PAS field (Video has access to
                     ; palette)

; ===== set VGA palette 0-15 to black =====
   MOV   DX,003C8h   ; VGA Palette Address Register
   MOV   AL,0        ; Set palette starting with color 0
   OUT   DX,AL       ; Set color 0.
   INC   DX          ; DX=003C9 (VGA Palette Data register)
   PUSH  DS          ; Save DS
   MOV   SI,Seg XBIN.PAL ; Segment of XBIN Palette in SI
   MOV   DS,SI       ; And copy in DS
   MOV   SI,Offset XBIN.PAL; Offset of XBIN Palette in SI
   MOV   CX,3*16     ; 16 times 3 values (Red, Green Blue)
   CLD               ;    Increment with LODS/STOS/MOVS
NxtPalVGA:
   LODSB             ; Load byte at DS:SI, increment SI
   OUT   DX,AL       ; Set R, G or B value.
   LOOP  NxtPalVGA   ; Loop for all 16*3 values.
   POP   DS          ; Restore DS
```

Setting the VGA in (non-)Blink mode.

When in blink-mode, the highest bit of the attribute byte determines whether the character will blink or not. In nonblink-mode this same bit selects high-intensity colors. Note however that 'high-intensity' does not really apply to XBin, since all of the 16 colors can be redefined. In any case, it's a matter of choosing between 16 background colors, or 8 background colors plus blinking.

Setting blink or nonblink mode is pretty easy really, all that needs to be done is set or clear a single bit in the 'Mode Control' register.

Enabling (default) blink mode:

```
   MOV   DX,003DAh   ; CRTC Status register
   IN    AL,DX       ; Read CRTC Status. (This will reset
                     ; the Attribute controller flip-flop)
; ===== Set blink bit =====
   MOV   DX,003C0h   ; Attribute controller (Write port)
   MOV   AL,10h+20h  ; Register 10h (Mode control)
                     ; leave PAS field enabled.
   OUT   DX,AL       ; Activate register 10h
   INC   DX          ; DX=003C1h (Attribute READ port)
   IN    AL,DX       ; Read Mode control register
   DEC   DX          ; DX=003C0h (Attribute Write port)
   OR    AL,008h     ; Set blink bit
   OUT   DX,AL       ; Rewrite Mode control register
```

Disabling blink mode:

```
   MOV   DX,003DAh   ; CRTC Status register
   IN    AL,DX       ; Read CRTC Status. (This will reset
                     ; the Attribute controller flip-flop)
; ===== Set blink bit =====
   MOV   DX,003C0h   ; Attribute controller (Write port)
   MOV   AL,10h+20h  ; Register 10h (Mode control)
```

```
                    ; leave PAS field enabled.
OUT   DX,AL         ; Activate register 10h
INC   DX            ; DX=003C1h (Attribute READ port)
IN    AL,DX         ; Read Mode control register
DEC   DX            ; DX=003C0h (Attribute Write port)
AND   AL,NOT 008h ; Clear blink bit
OUT   DX,AL         ; Rewrite Mode control register
```

! NOTE ! When running in Windows in windowed mode, whatever you try you **always** have nonblink mode. Switching back and forth between full-screen mode and windowed mode, can have effect on the blink/nonblink status. In case you're wondering... it's just one of the shitload of bugs in Windows.

Setting the VGA in 512 Character mode.

When in 512 character mode, bit 3 of the attribute byte selects which of the two character sets to use. This does mean of course that you can only have a maximum of 8 foreground colors.

It would be VERY tedious to draw an image in 512 character mode without a program to support it, and this is probably the reason why this feature is almost never used. Then again, it could be quite usefull for certain applications. Since there is no 'predefined' set of characters for the extra set of characters, you could use it to have special 'pictures' and 'icons' in textmode without it ever interfering with the actual 256 character ASCII set.

Setting up the VGA in 512 character mode is pretty easy. The "character map select" register holds which two out of 8 possible loaded character sets you want to select. When both the values for character set A, and character set B are identical, 256 character mode is selected, when they differ, 512 character mode is selected... Gee, THAT's easy ain't it ? :-)

Since we basically want to have the easiest way to have it appear to have one continuous set of 512 characters, all we need to do is program the Character Map Select register so we have the first 256 characters in the TOP 8Kb of font memory, and the second 256 characters in the NEXT 8K of font memory. To do this we must set Character set A to 0, and Character set B to 4, the reason for this 0 and 4, in stead of the more logical 0 and 1 is to provide backwards compatibility with the EGA.

Enabling 512 character mode:

```
MOV   DX,003C4h   ; Sequencer register
MOV   AX,01003h   ; Character Map Select in AL (index 3)
                  ; And setup for 4 in SBH field and 0
                  ; in SAH field loaded in AH
OUT   DX,AX       ; And write Value.
```

Disabling 512 character mode (enabling 256 character mode)

```
MOV   DX,003C4h   ; Sequencer register
MOV   AX,00003h   ; Character Map Select in AL (index 3)
                  ; And setup for 0 in SBH field and 0
                  ; in SAH field loaded in AH
OUT   DX,AX       ; And write Value.
```

Simple eh ? All that's left now is to actually load the font, and setting font size. We'll come to that part shortly.

Blinking and 512 Characters.

As a reminder. The attribute byte is set up as follows:

Attribute Byte layout

7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
B/I	Red	Green	Blue	I/512	Red	Green	Blue

- Bit 0: Blue foreground
- Bit 1: Green foreground
- Bit 2: Red foreground
- Bit 3: Intensity **AND** selects first (when 0) or second (when 1) set of 256 characters in 512 character mode.
- Bit 4: Blue background
- Bit 5: Green background
- Bit 6: Red background
- Bit 7: Blinking or Intensity (when in non-blink mode).

Please note, the R, G, and B values are assuming the DEFAULT palette, when you reprogram the palette there is really no longer any Red, Green and Blue value as described here. Heck, you could have ONLY shades of red for all that matter.

In the same way, the 'Intensity' not necesarrily means 'Brighter' color. Setting up color value 4 to blue, and color value 12 ('Intensity' color 4) to green, would have it blinking between blue and green.

When 512 character mode is enabled, Bit 3 determines both which of the two 256 character sets to use **AND** the intensity bit. This can lead to confusing results in a XBin drawing package. A simple way to work around this is to program the palette so that the first and last eight palette values are identical.

Setting font size.

The default VGA text mode is a 720*400 pixel mode. Each character is made up of a matrix of 8 pixels wide and 16 pixels high. Something strange happens here. When IBM developed the VGA, one of it's intensions was to provide a more readable and 'nicer' looking character set compared to the EGA which utilized a 8*14 font. The main problem however was not having a character matrix with more lines, but a character matrix with more columns, since a PC typically works with BYTES however, it'd be pretty hard to organize a system with for example 9, 10 or 11 bits per scanline. So they adopted for the system already widely in use on the Hercules card.

The Hercules provided a better character by having 9 pixels per font row, however, NO extra bit was needed. As it turns out, the only problem is the small space in between to adjacent characters... Why not utilize 8 bits from a byte to have an 8 pixel wide actual font, and insert a blank pixel so there's some space... And this is exactly how it works.

There are a few exceptions though, the box drawing characters would look messy if they were separated by one pixel so for those characters, the eigth column is doubled. There is NO way to select which of the 256/512 characters should have a blank column or column 8 doubled, the set of characters with this doubling is fixed they are 0C0h to 0DFh (192 to 223). You CAN however disable column doubling altogether or switch the VGA into 640*400 mode, so it utilises an 8 pixel wide font. This does reduce the quality of the characters slightly, but this usually not a bad thing, and ANSi's tend to look prettier in 640*400. ;-)

Switching the VGA into 8 pixel mode involves some pretty messy bit twiddling. You should generally set 8 pixelmode immediately after setting the videomode. This code WILL most likely cause the screen to flicker. On a multisyncing or variable scanning rate monitor, it will cause the monitor to go blank for a short period of time while the monitor adjusts to the new scanning rate. It will generally cause the same sort of flicker/disturbance as when you switch from textmode to graphics and/or back. Since the setting a video mode will probably already do the same, it will cause the least amount of disturbance right after a mode set. I'm not providing any code for switching back from 8 to 9 pixel font, I recommend switching back to textmode using BIOS interrupt 10, function 0, set mode. If you really need code to switch back to 9 pixel mode, you will need to derive the appropriate method from the following code.

Setting 640 pixel wide screen (and a 8 pixel wide characterbox).

```
MOV   DX,003CCh   ; Misc output register READ port
IN    AL,DX       ; Read value.
AND   AL,0F3h     ; Turn off Bits 2&3 (Clock select 0).
MOV   DX,003C2h   ; Misc Output Write port
OUT   DX,AL       ; Writeback modified value
CLI               ; NO interrupts for a while
MOV   DX,03C4h    ; Sequencer register
MOV   AX,100h     ; \ Generate and hold Synchronous reset
OUT   DX,AX       ; /
MOV   AL,001h     ; Clocking mode register
OUT   DX,AL       ; Activate Clocking mode register
INC   DX          ; Data register
IN    AL,DX       ; Read value
OR    AL,1        ; Set Bit 0 (8/9)
OUT   DX,AL       ; Writeback.
DEC   DX          ; Back to Address register
MOV   AX,300h     ; \ Release Reset state. (normal)
OUT   DX,AX       ; /
STI               ; Interrupts allowed again
```

Even though the VGA allows to have 6, 8, 12 and 16 pixel wide actual fonts, I'm not going to go any further on that here, XBin only supports 8 pixel wide fonts anyway.

XBin does allow for setting any font **height** from 1 to 32 however, so we need to be able to set that. As it turns out, this too is pretty simple, all we need to do it set the "Maximum Scanline Register" to the number of required scanlines minus one,

```
MOV   DX,003D4h   ; CRTC address register
MOV   AL,9        ; Index for Maximum Scanline Register
OUT   DX,AL       ; set MSL as active register
INC   DX          ; Set DX to CRTC Data register
IN    AL,DX       ; read current MSL
AND   AL,011100000b ; set MSL field to 0, preserve others
MOV   AH,XBIN.Fontsize ; get fontsize from XBIN header
DEC   AH          ; minus one.
OR    AL,AH       ; set size in MSL field
OUT   DX,AL       ; Writeback modified value
```

Setting a characterset (font).

The VGA allows you to load up to 8 fonts into fontmemory, and then using the Character Map Select register, select which of the eight you currently want to use. There's too much to tell about all the possibilities here, so we'll focus on the possible needs by XBin.

When in textmode, the VGA really works in a pretty odd way. The memory is organised into four planes just as with 16 color graphics. Character bytes reside on plane 0, attribute bytes reside on plane 1, and the font is available in plane three. To provide compatibility with CGA & EGA, planes 0 and 1 are also mapped in the B800h address space. It's a little off topic to try to explain all of it, heck it takes an entire chapter in most VGA reference books, we'll focus only on the needs of XBin.

In the way the Character Map Select was setup earlier, all we need to do is load a 256 character font into the first 8Kb of fontmemory, or a 512 character font into the first 16Kb of font memory.

Because of the character/attribute mapping to 0B800h, the video memory is no longer available as a linear set of memory. Setting a font will consist of three major blocks.

1. Initialize the VGA for reading/writing font data.
    - Setting the memory to "Sequential" addressing mode.
    - Select memory plane 2 for read and write access.
    - Disable odd addressing mode.
    - Select 0A000h as memory range in stead of 0B800h.
2. Actually loading the font.
3. Restoring the VGA to it's original state.
    - Setting memory to "Odd/Even" addressing mode.
    - Select plane 0 and 1 for write access.
    - Select plane 0 for read access.
    - Enable odd addressing mode.

Steps 1 and 3 involve accessing VGA registers, Step 2 is really pretty straightforward, and merely consists of copying the font data into font memory in the appropriate format required by the VGA (more on this in a moment).

Initialize the VGA for reading/writing font data

```
CLI             ; No interrupts allowed
MOV   DX,003C4h ; Sequencer register
MOV   AX,0100h  ; \ Synchronous reset
OUT   DX,AX     ; /
MOV   AX,0402h  ; \ Select Plane 2 for WRITE
OUT   DX,AX     ; /
MOV   AX,0704h  ; \ Sequential Addressing mode
OUT   DX,AX     ; /
MOV   AX,0300h  ; \ Release Synchronous reset
OUT   DX,AX     ; /
MOV   DX,003CEh ; Graphics controller register
MOV   AX,0204h  ; \ Select Plane 2 for READ
OUT   DX,AX     ; /
MOV   AX,0005h  ; \ Disable odd addressing mode
OUT   DX,AX     ; /
MOV   AX,0006h  ; \ Memory range is A000:0000
OUT   DX,AX     ; /
STI             ; Interrupts enabled
```

Restoring the VGA to it's original state

```
CLI             ; No interrupts allowed.
MOV   DX,003C4  ; Sequence controller register
MOV   AX,0100h  ; \ Synchronous reset
OUT   DX,AX     ; /
MOV   AX,0302h  ; \ Select Plane 0 & 1 for WRITE
```

```
   OUT   DX,AX       ; /
   MOV   AX,0304h    ; \ Odd/Even Addressing mode
   OUT   DX,AX       ; /
   MOV   AX,0300h    ; \ Release Synchronous reset
   OUT   DX,AX       ; /
   MOV   DX,003CEh   ; Graphics controller register
   MOV   AX,0004h    ; \ Select Plane 0 for READ
   OUT   DX,AX       ; /
   MOV   AX,1005h    ; \ Enable odd addressing mode
   OUT   DX,AX       ; /
   MOV   AX,0E06h    ; \ Memory range is B800:0000
   OUT   DX,AX       ; /
   STI               ; Interrupts enabled
```

Loading a font Fontmemory is organised in an array of 256 (or 512) character matrices. Each matrix is exactly 32bytes in size. a 256 character font thus occupies 8Kb of memory, nomatter what the 'height' of the font. An XBin font however does not enforce this 32bytes size. While the VGA had a speed consideration, XBin has a size consideration, it would waste too much space. It's easy to convert the XBin format to VGA format, all that needs to be don is insert some gaps in between each character. The following does just that.

```
   PUSH  DS
   MOV   DI,0A000h
   MOV   ES,DI
   XOR   DI,DI       ; point ES:DI to screen font (A000:0000)
   LDS   SI,[FontData]; DS:SI points to font data
   MOV   BX,256      ; (or 512) number of characters
   MOV   AX,[CharSize]; Size of character
   CLD               ; Increment on MOVSB, STOSB...
   MOV   DX,32       ; \ DX=32-CharSize (Size of gap in
   SUB   DX,AX       ; / between two screen memory chars)
NextChar:
   MOV   CX,AX       ; CX=FontHeight
   REP   MOVSB       ; Copy 'FontHeight' bytes to videomem
   ADD   DI,DX       ; Skip to next character (leave a gap)
   DEC   BX          ; One character done.
   JNZ   @NextChar
   POP   DS
```

Decompressing an XBin.

Decompressing an XBin is pretty straightforward. First you would need to allocate an array of the required size, and then decompress the XBin data into the array. Depending on how rugged you want to make it, checks for invalid XBin files are needed. Then again, you could just depend on the XBin on being without error, and have your program crash when it's not ;-)

The decompression process is very simple. Load a counter/compression type byte and depending on the compression type decode the correct number of bytes.

A rough outline (no error checks). For size reasons, I've deliberately not used REAL usable code, It should be fairly easy to create that yourself. An actual working routine is available in the SimpleXB viewer in the XBin Developers Kit.

```
WHILE Still_bytes_in_XBIN
   Counter := Character_from_XBIN
   Type    := Counter & 0C0h ; Mask out bits 6 and 7
   Counter := Counter & 03Fh ; Remore 'Type' bits from Counter.
```

```
   IF (Type = 000h) THEN     ; No compression
      FOR i:=1 TO Counter+1
         Character := Character_from_XBIN
         Attribute := Character_from_XBIN
         Store_In_Array (Character, Attribute)
      ENDFOR
   ELSEIF (TYPE = 040h) THEN ; Character compression
      Character := Character_from_XBIN
      FOR i:=1 TO Counter+1
         Attribute := Character_from_XBIN
         Store_In_Array (Character, Attribute)
      ENDFOR
   ELSEIF (TYPE = 080h) THEN ; Attribute compression
      Attribute := Character_from_XBIN
      FOR i:=1 TO Counter+1
         Character := Character_from_XBIN
         Store_In_Array (Character, Attribute)
      ENDFOR
   ELSE                      ; Character/Attribute compression
      Character := Character_from_XBIN
      Attribute := Character_from_XBIN
      FOR i:=1 TO Counter+1
         Store_In_Array (Character, Attribute)
      ENDFOR
   ENDIF
ENDWHILE
```

On-the-fly (realtime) decompression.

In stead of decompressing an entire XBin, and then use the decompressed data to display the XBin, you could create a system where you only decompress the part of the XBin you actually need. While being more complex, it **does** allow for a more memory efficient viewer capable of viewing XBin files substantially larger than available memory. While not impossible, it'd probably be too difficult to create an ANSi/XBin editor making full use of this possibility. The provided viewer SimpleXB.EXE/SimpleXB.PAS in the XBin Developers Kit uses this method, making it possible to view XBin files of any size (even of 65535 by 65535 in size !) Note that this **does** make the viewer slower, since it continuously needs to load portions of the XBin from disk. Better ways for having realtime decompression are possible, but it would make the **simple** viewer too **complex** ;-).

Compressing an XBIN.

This is somewhat harder to explain... While it looks easy at first, it takes quite a bit of work to create an XBin compressor that achieves good to perfect compression. The Sample BIN2XBIN (in the XBin Developer kit) program, includes a complete and near-perfect compression algorythm.

As a test case I've also created an optimal XBin compressor that makes the smallest possible XBin, it is however considerable more complex, a magnitude slower and consumes a massive amount more memory than the provided BIN2XBIN program. It basically tests all possible compression types at each character, reverting to the smallest size it found in the process. Several tests on compressing the ANSi's in a dozen or so art packs from the last few months indicated the compression algorythm in BIN2XBIN provides very good compression at very little memory and time cost. None of the ANSi's I tried (which I converted to BIN first) resulted in a XBin that was bigger as it's BIN. It is however possible to create a worst-case situation, in which the BIN2XBIN consitently makes the wrong decisions resulting in a file being 33.33% bigger than the optimal compressed one. Luckily, those

ANSi's tend to be very unattractive by nature. I've spent quite a while thinking about how to implement a fast, yet simple and efficient XBIN compressor. Nevertheless, nobody is perfect. If you do find an algorythm that consistently provides better compression than BIN2XBIN I would like to hear from you. Actual code is available in BIN2XBIN.PAS.

Back to XBIN

This page is being maintained by

Tasmaniac@acid.org