VESA®

Video Electronics Standards Association

2150 North First Street, Suite 440 Phone: (408) 435-0333
San Jose, CA 95131-2029 FAX: (408) 435-8225

VESA BIOS EXTENSION (VBE)
Core Functions
Standard

Version: 3.0

Date: September 16, 1998

Purpose

To standardize a modular, software interface to display and audio devices. The VBE interface is intended to simplify and encourage the development of applications that wish to use graphics, video, and audio devices without specific knowledge of the internal operation of the evolving target hardware.

Summary

The VBE standard defines a set of extensions to the VGA ROM BIOS services. These functions can be accessed under DOS through interrupt 10h, or be called directly by high performance 32-bit applications and operating systems other than DOS.

These extensions also provide a hardware-independent mechanism to obtain vendor information, and serve as an extensible foundation for OEMs and VESA to facilitate rapid software support of emerging hardware technology without sacrificing backwards compatibility.

This page is intentionally blank.

Intellectual Property

Copyright © 1993-1998 - Video Electronics Standards Association. All rights reserved.

While every precaution has been taken in the preparation of this standard, the Video Electronics Standards Association and its contributors assume no responsibility for errors or omissions, and make no warranties, expressed or implied, of functionality or suitability for any purpose.

The sample code contained within this standard may be used without restriction.

Trademarks

All trademarks used in this document are property of their respective owners.

- VESA, VBE, VESA DDC, VBE/AI, VBE/PM, and VBE/DDC are trademarks of Video Electronics Standards Association.
- MS-DOS and Windows are trademarks of Microsoft , Inc.
- IBM, VGA, EGA, CGA, and MDA are trademarks of International Business Machines
- RAMDAC is a trademark of Brooktree Corp.
- Hercules is a trademark of Hercules Computer Technology, Inc.

Patents

VESA proposal and standards documents are adopted by the Video Electronics Standards Association without regard to whether their adoption may involve patents on articles, materials, or processes. Such adoption does not assume any liability to any patent owner, nor does it assume any obligation whatever to parties adopting the proposal or standards document.

Support for this Specification

Clarifications and application notes to support this standard will be published as the need arises. To obtain the latest standard and support documentation, contact VESA.

If you have a product which incorporates VBE, you should ask the company that manufactured your product for assistance. If you are a display or controller manufacturer, VESA can assist you with any clarification you may require. All comments or reported errors should be submitted in writing to VESA, to the attention of Technical Support, using one of the following mechanisms:

World Wide Web: www.vesa.org
E-mail: support@vesa.org
Fax: 408-435-8225
Voice: 408-435-0333

Mail to:
Video Electronics Standards Association
2150 North First Street, Suite 440
San Jose, California 95131-2029

SSC VBE/Core Workgroup Members

Any industry standard requires input from many sources. The people listed below were members of the VBE/Core Workgroup of the Software Standards Committee (SSC) which was responsible for combining all of the industry input into this standard:

CHAIRMAN

Charimain, David Penley, Cirrus Logic, Inc.
past chairnam, Kevin Gillett, S-MOS Systems, Inc.,
past chairman, Rod Dewell , Excalibur Solutions

MEMBERS

Jatinder Pancar, Alliance Seminconductor
Armond Bruno, BrookTree Corporation
Rebecca Nolan, Chips and Technologies, Inc.
Andy Sobczyk, Cirrus Logic, Inc.
Brad Haakenson, Cirrus Logic, Inc.
Adrian Luff, Forte Technologies, Inc.
Steven McGowen, Intel Corporation
Jake Richter, Jon Peddie and Associates
Matt Russo, Matrox Graphics, Inc.
Todd Laney, Microsoft Corporation
Thomas Block, Number Nine Visual Technology Corporation
Mark Krueger, NVidia Corporation
Dwight Diericks, NVidia Corporation
Rex Wolfe, Phillips Semiconductor
Raluca Iovan, Phoenix Technologies Ltd.
Tim Crawford, Rendition, Inc.
Kendall Bennett, SciTech Software, Inc
Tom Ryan, SciTech Software, Inc
Don Pannell, Sierra Semiconductor
George Bystricky, S-MOS Systems, Inc.
David Milici, StereoGraphics Corporation
Tony Lin, Trident Microsystems, Inc.
Mitch Paris, Tseng Labs, Inc.
Joe Israel, Tseng Labs, Inc.
Chris Tsang, ULSI Systems
Danny Halamish, VideoLogic, Inc..
Gregory Hamlin, VRex, Inc.
Thomas Roell, X Inside Inc.

Table of Contents

Introduction

This document contains the VESA BIOS Extension (VBE) specification for standard software access to graphics display controllers which support resolutions, color depths, and frame buffer organizations beyond the VGA hardware standard. It is intended for use by both applications programmers and system software developers. It is also intended to provide an extended interface to support enhanced refresh rate control for operating system utilities and drivers.

System software developers may use this document to supplement the System and INT 10h ROM BIOS functions to provide the VBE services. Application developers can use this document as a guide to programming all VBE compatible devices.

To understand the VBE specification, some knowledge of 80x86 assembly language and the VGA hardware registers may be required. However, the services described in this specification may be called from any high-level programming language that provides a mechanism for generating software interrupts with the 80x86 registers set to user-specified values.

In this specification, 'VBE' and 'VBE 3.0' are synonymous with 'VBE Core Functions version 3.0'.

Scope of the VBE Standard

The primary purpose of the VESA VBE is to provide standard software support for the many unique implementations of Super VGA (SVGA) graphics controllers on the PC platform that provide features beyond the original VGA hardware standard. This is to provide a feasible mechanism by which application developers can take advantage of this nonstandard hardware in graphics applications.

The VBE specification offers an extensible software foundation which allows it to evolve as display and audio devices evolve over time, without sacrificing backward software compatibility with older implementations. New application software should be able to work with older hardware, and application software that has already shipped should work correctly on new hardware devices.

VBE services provide standard access to all resolutions and color depths provided on the display controller, and report the availability and details of all supported configurations to the application as necessary.

VBE implementations facilitate the field support of audio and display hardware by providing the application software with the manufacturer's name and the product identification of the display hardware.

Since graphics controller services on the PC are typically implemented in ROM, the VBE services are defined so that they should be implemented within the standard VGA ROM. When ROM

implementations of VBE are not possible, or when field software upgrades to the onboard ROM are necessary, the VBE implementation may be also offered as a device driver or DOS Terminate and Stay Resident (TSR) program.

The standard VBE functions may be supplemented by OEM's as necessary to support custom or proprietary functions unique to the manufacturer. This mechanism enables the OEM to establish functions that may be standard to the product line, or provide access to special hardware enhancements.

Although previous VBE standards assumed that the underlying graphics architecture was a VGA device, the display services described by VBE 3.0 can be implemented on any frame buffer oriented graphics device.

The majority of VBE services facilitate the setup and configuration of the hardware, allowing applications high performance, direct access to the configured device at runtime. To further improve the performance of flat frame buffer display devices in extended resolutions, VBE 3.0 provides new memory models that do not require the traditional frame buffer "banking" mechanisms.

VBE is expected to work on all 80x86 platforms, in real and protected modes. Starting with VBE 3.0, all the VBE/Core BIOS functions can be ‘dual-mode’, allowing them to optionally be called as 16-bit protected mode code via a direct call to a new protected mode interface entry point. ‘Dual-mode’ code means that the BIOS code adheres to certain restrictions when called via the protected mode entry point, to ensure full compatibility with fully protected mode operating systems such as Windows NT, OS/2 and the many versions of UNIX. Note that although the ‘dual-mode’ code must be called as 16-bit protected mode code, this does not preclude it from being called by 32-bit pure operating systems such as Windows NT and OS/2. Since some modern display devices are designed without any VGA support, two or more display controllers may be present in the system. One display controller could be used for VGA compatibility, and the other used for graphic extensions to the basic VGA modes, resolutions, and frame buffer models. Since it is not possible to support multiple controllers easily via the INT 10h software interface, only the primary controller will be supported via this interface and its BIOS will be located at C0000h. If multiple controllers are present in the system, the second controller can only be controlled via the VBE/AF Accelerator Functions specification (contact VESA for more information).

Note that the VBE/Core specification does not include any support for hardware acceleration functions such as 2D and 3D graphics primitives or video acceleration. If you wish to use such features please refer to the VBE/AF Accelerator Functions specification (contact VESA for more information).

Backgrounder

The IBM VGA[1] has become a de facto standard in the PC graphics world. A multitude of different VGA offerings exist in the marketplace, each one providing BIOS or register compatibility with the IBM VGA. More and more of these VGA compatible products implement various supersets of the VGA standard. These extensions range from higher resolutions and more colors to improved performance and even some graphics processing capabilities. Intense competition has dramatically improved the price/performance ratio, to the benefit of the end user.

However, several serious problems face a software developer who intends to take advantage of these "Super VGA"[2] environments. Because there is no standard hardware implementation, the developer is faced with widely different Super VGA hardware architecture. Lacking a common software interface, designing applications for these environments is costly and technically difficult. Except for applications supported by OEM-specific display drivers, very few software packages can take advantage of the power and capabilities of Super VGA products.

The VBE standard was originally conceived to enable the development of applications that wished to take advantage of display resolutions and color depths beyond the VGA definition. The need for an application or software standard was recognized by the developers of graphic hardware to encourage the use and acceptance of their rapidly advancing product families. It became obvious that the majority of software application developers did not have the resources to develop and support custom device level software for the hundreds of display boards on the market. Therefore the rich new features of these display devices were not being used outside of the relatively small CAD market, and only then after considerable effort.

Indeed, the need for a standard for SVGA display adapters became so important that the VESA organization was formed to seek out a solution. The original VBE standard was devised and agreed upon by each of the active display controller manufacturers, and has since been adopted by DOS application developers to enable use of non-VGA extended display modes.

As time went along VBE 1.1 was created to add more video modes and increased logical line length/double buffering support. VBE 1.2 was created to add modes and also added high color RAMDAC support.

In the three years since VBE 1.2 was approved we have seen the standard become widely accepted and many successful programs have embraced VBE. However, it has become obvious that the need for a more robust and extensible standard exists. Early extensions to the VGA standard continued using all of the original VGA I/O ports and frame buffer address to

[1] IBM and VGA are trademarks of International Business Machines Corporation.

[2] The term "Super VGA" is used in this document for a graphics display controller implementing any superset of the standard IBM VGA display adapter.

communicate with the controller hardware. As we've seen, the supported resolutions and color depths have grown, intelligent controllers with BITBLT and LineDraw Functions have become common, and new flat frame buffer memory models have appeared along with display controllers that are not based on VGA in any way. VBE 2.0 and future extensions will support non-VGA based controllers with new functions for reading and writing the palette and for access to the flat frame buffer models.

VBE 3.0, as designed, offers the extensibility and the robustness that was lacking in the previous specifications, while at the same time offering backwards compatibility.

VBE Overview

This chapter outlines the various features and limitations of the VBE 3.0 standard.

VBE Features

- Standard application interface to Graphics Controllers (SVGA Devices).
- Optional protected mode interface for OS's such as Windows NT, OS/2 and UNIX.
- Standard method of overriding the refresh rate for supported modes.
- Stereoscopic display support for liquid crystal (LC) shutter glasses.
- Standard method of identifying products and manufacturers.
- Provision for OEM extensions through Subfunction 14h.
- Extensible interface through supplemental specifications.

VBE Affected Devices

All frame buffer-based devices in the PC platform (with the exception of Hercules, Monochrome (MDA), CGA and EGA devices) are suitable for use within the VBE standard to enable access to the device by VBE-compliant applications.

Providing Vendor Information

The VGA specification does not provide a standard mechanism to determine what graphic device it is running on. Only by knowing OEM-specific features can an application determine the presence of a particular graphics controller or display board. This often involves reading and testing registers located at I/O addresses unique to each OEM. By not knowing what hardware an application is running on, few, if any, of the extended features of this hardware can be used.

The VESA BIOS Extension provides several functions to return information about the graphics environment. These functions return system level information as well as graphics mode specific details. Function 00h returns general system level information, including an OEM identification string. The function also returns a pointer to the supported VBE and OEM modes. Function 01h may be used by an application to obtain additional information about each supported mode. Function 03h returns the current VBE mode.

Programming with VBE/Core

This section contains application and systems programming information for some of the more advanced functions that VBE 3.0 provides.

Accessing Linear Framebuffer Memory

Once you have initialized the graphics hardware into a mode that supports a hardware linear framebuffer, you need to create a pointer that your application can use to write to the linear framebuffer memory. The first thing you should realize is that the linear framebuffer location reported in the ModeInfoBlock for the mode you are using is a *physical* memory address, and cannot be used directly from protected mode applications. Before you can use the memory you must use the services your operating system provides to map the physical memory to a linear memory address, and then map this linear address into your applications memory space. Under DPMI mapping the linear memory is done with DPMI function 0x800, and equivalent functions exist under other operating systems.

The steps involved in mapping in a linear framebuffer region are as follows (32-bit protected mode only):

1. Map the physical memory address to a linear memory address (using DPMI function 0x800 for example).

2. Find the base address of the default DS selector for your operating environment.

3. Subtract the base address from the linear address computed in step 1 to give you a near pointer (relative to DS) that you can use from within your code.

Using Refresh Rate Control

VBE 3.0 provides support for refresh rate control by allowing the calling application to pass a set of custom CRTC timing values to the BIOS when a mode set is being performed. This provides for maximum versatility and allows the application to program specific CRTC timing values if this is necessary (for instance specialized display hardware such as head mounted displays or fixed frequency monitors).

When the calling application wishes to control the refresh rate for the mode being initialized, it must compute a set of CRTC values and a normalized pixel clock that can be passed to function 4F02h when the mode is initialized. The VBE 3.0 interface does not provide any means to compute these values, and the values can either be taken from discrete VESA DMT timings, or by using the new VM/GTF Generalized Timing Formula to compute the CRTC values to be used. Once you have these values, you must search for an available pixel clock that is the closest to

what you want by calling VBE function 4F0Bh (if you are using GTF to compute the CRTC values, you should re-run the GTF calculations routines based on the pixel clock returned by 4F0Bh to get the final CRTC values).

The steps involved in initializing a mode with a specific refresh rate are as follows:

1. Use VBE/DDC to obtain the operational limits of the monitor if VBE/DDC is available and the monitor is DDC compliant. Use these values to restrict the calculated values or modes that you can make available.

2. Use either the DMT timings or GTF formulas to compute a set of CRTC timings and normalized pixel clock for the mode and refresh rate that you want to initialize. If the mode is a double scanned mode (200, 240 or 300 lines) double the vertical resolution used to drive the GTF CRTC calculations (i.e.: use 400 lines for the vertical number of lines).

3. Call VBE function 4F0Bh to find the actual pixel clock that will be programmed for the normalized pixel clock that you have requested.

4. If you are using GTF, re-run the GTF formulas using the resulting normalized pixel clock returned by function 4F0Bh to compute the proper CRTC timings given the exact pixel clock that will be used for the resulting mode. This is an important step to ensure that the GTF timings will be exact for the resulting mode given the available pixel clocks the hardware supports (and PLL resolution for programmable pixel clocks).

5. If you are setting a double scanned mode (200, 240 or 300 lines vertical) set the double scan flag in the CRTCInfoBlock.

6. If you are enabling an interlaced mode, set the interlace flag in the CRTCInfoBlock.

7. Call function 4F02h with bit D11 set in the mode number and pass in the CRTC timings and pixel clock in the CRTCInfoBlock.

Using VBE/DDC to obtain monitor operational limits

The VBE/DDC interface can be used to obtain the operational limits of the attached monitor, such as the minimum and maximum horizontal and vertical frequencies as well as supported resolutions. If the monitor and graphics card both support DDC, this information should be obtained and used to restrict the refresh rate computation routines to ensure that the resulting CRTC values do not produce a mode that lies outside the operational limits of the attached monitor.

Please refer to the VBE/DDC and EDID specifications for more information on how to obtain the operational limits of the monitor.

Using VM/GTF to compute CRTC values

The VESA Monitor Committee's GTF (Generalized Timing Formula) standard defines a set of formulas that can be used to compute GTF compatible CRTC timings given a couple of input parameters. GTF can be used to generate CRTC timings given the resolution and one of either the vertical frequency, horizontal frequency or pixel clock. For more information please refer to the GTF specification for the formulas and sample code for calculating GTF compliant CRTC values.

Note that GTF can be used to generate timings that will display properly on both new GTF compatible monitors and for existing non-GTF compliant monitors.

Computing the normalized pixel clock

Once you have generated a set of CRTC timings from the DMT timings or using the GTF formula, the pixel clock will be an arbitrary pixel clock which may not be directly programmable in the hardware. Once you have the normalized pixel clock, you must then call VBE function 4F0Bh to find the exact value of the closest pixel clock that the hardware can program. Note that when you call function 4F0Bh, you *must* pass it the correct value of the graphics mode that you will be using the pixel clock for. This allows the underlying VBE implementation to make any necessary adjustments before running the clock through the PLL calculation routines so that you will be returned the exact pixel clock that will be programmed by the hardware.

Setting double scan modes

If double scanning is supported by the hardware, it is possible to support modes like 320x200, 320x240 and 400x300 that scan each line twice in the vertical direction. These modes cannot be implemented without double scanning support, and if the application is attempting to perform refresh rate control on these modes, setting the double scan bit is required.

When setting a double scan mode, the actual CRTC parameters calculated and passed in will be equal to double the number of vertical lines. Hence for a 200 line mode you would actually set the vertical CRTC parameters and pixel clock for a 400 line mode, and set the double scan bit to convert the mode to a 200 line addressable mode. Note that if you are using double scanned modes, you cannot enable interlaced operation.

Setting Interlaced Modes

If interlaced mode is supported by the hardware, bit D9 will be set in the ModeAttributes field of the ModeInfoBlock for the graphics mode in question. If this bit is set, you can enable interlaced operation by setting bit D1 of the Flags field of the CRTCInfoBlock passed in to function 4F02h. The CRTC timings for the interlaced mode will be identical to the non-interlaced mode.

Note that some new hardware may not support interlaced operation, so make sure you check the ModeInfoBlock attributes field to ensure interlaced is supported before attempting to initialize an interlaced mode. Note that if you are using double scanned modes, you cannot enable interlaced operation. Generally interlaced operation is only available for modes with resolutions of 640x480 and higher, and most hardware will have a hard time enabling interlaced operation for lower resolution modes.

Using Hardware Triple Buffering

Hardware triple buffering is supported in the VBE/Core specification by allowing the application to schedule a display start address change, and then providing a function that can be called to determine the status of the last scheduled display start address change. VBE Function 4F07h is used for this, along with subfunctions 02h and 04h. To implement hardware triple buffering you would use the following steps:

1. Display from the first visible buffer, and render to the second hidden buffer.

2. Schedule a display start address change for the second hidden buffer that you just finished rendering (4F07h, 02h), and start rendering immediately to the third hidden buffer. CRT controller is currently displaying from the first buffer.

3. Before scheduling the display start address change for the third hidden buffer, wait until the last scheduled change has occurred (4F07h, 04h returns not 0). Schedule display start address change for the third hidden buffer and immediately begin rendering to the first hidden buffer. CRT controller is currently displaying from the second buffer.

4. Repeat step 3 over and over cycling though each of the buffers.

Although the above method does require a spin loop polling on Function 4F07h/04h, in most cases when this function is called the display start address change will already have occurred and the spin loop will time out immediately. The only time that this cannot occur is if the application is rendering at a frame rate in excess of the current hardware refresh rate (i.e.: in excess of 60-70 frames per second), and the resulting frame rate for the application will be pegged at the hardware refresh rate.

Using Stereoscopic Liquid Crystal Shutter Glasses

Stereoscopic liquid crystal (LC) shutter glasses are a cheap, easy solution for getting real 3D stereoscopic imaging out of a standard PC with any standard monitor. LC shutter glasses work by constantly blanking out video information for each eye in a sequential fashion, allowing the user to see the left image for a fraction of a second followed by the right image, followed by the left image again etc. In order to make LC shutter glasses work effectively on PC based graphics controllers, some mechanism for changing the displayed video information at every vertical retrace is necessary. New hardware is available that will do this automatically, and VBE 3.0 defines the software interface necessary to allow applications to use these new hardware features.

There are two methods of performing hardware stereoscopic page flipping with VBE 3.0, depending on the available underlying hardware capabilities. All VBE implementations that have stereoscopic should support the first method, while the second method requires that the hardware implement dual display start addresses. Please check the mode information ModeAttributes field bit D12 to see if dual display start addressing is supported.

The steps involved in enabling free running stereoscopic support are as follows (Method 1 - consecutive left right images):

1. Set the mode via function 4F02h (using a high refresh rate if possible)

2. Enable free running stereoscopic mode with function 4F07h subfunction 05h.

3. Perform standard double buffered or triple buffered graphics using function 4F07h subfunctions 02h/04h/82h, but draw both the left and right images consecutively in memory rather than just a single image. Once both left and right eye images are rendered, the display can be swapped as would be done for normal double/triple buffered graphics.

4. Disable free running stereoscopic mode with function 4F07h subfunction 06h when you are done with stereoscopic viewing.

The steps involved in enabling free running stereoscopic support are as follows (Method 2 - dual display start addresses):

1. Set the mode via function 4F02h (using a high refresh rate if possible)

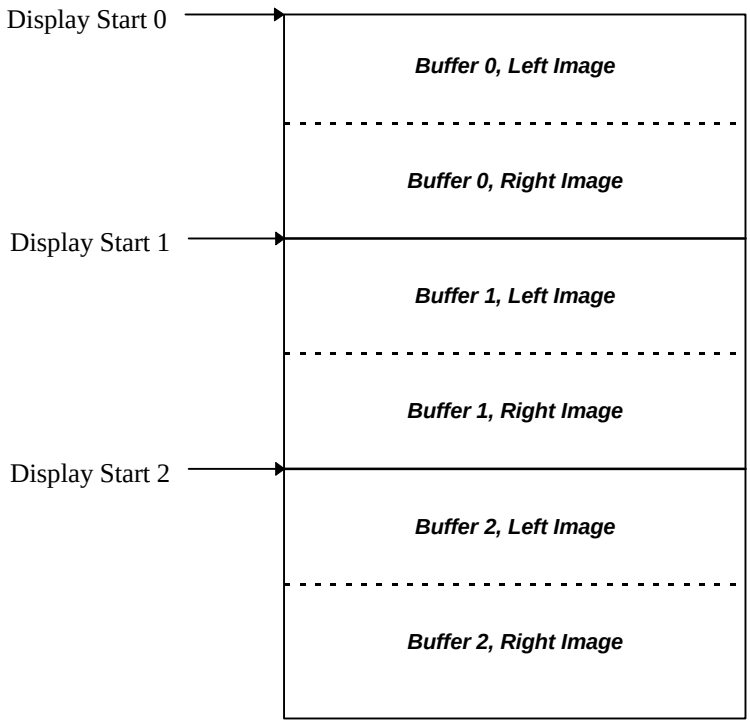
2. Draw the left and right images in video memory any way you wish (see below for some possible scenarios), and use subfunctions 03h or 83h to set the display start addresses for the left and right images in memory. Calling subfunction 03h or 83h will automatically enable free running stereoscopic mode for the controller. You can then render your next frame in memory and use 03h/04h/83h to perform standard double buffered or triple buffered graphics with both a left and right image for each buffer.

3. Disable free running stereoscopic mode with function 4F07h subfunction 06h when you are done with stereoscopic viewing.

Automatic hardware display start address swapping (Method 1)

VBE 3.0 supports hardware implementations that can be programmed to automatically swap between the left and right images of the stereoscopic display automatically every vertical retrace. Normally when a mode is set, stereoscopic mode will be disabled and can be enabled by the application by calling function 4F07h, subfunction 05h (and disabled with subfunction 06h). Once stereoscopic mode has been enabled, the hardware will display the left image from the data in video memory defined by the current display start address, followed by the right image in the following frame from the video memory immediately following the left image. After the right image has been displayed the hardware will begin displaying the left image again in a continuous cycle.

The diagram below shows how the pages would be laid out in video memory for a triple buffered, stereoscopic enabled graphics mode:

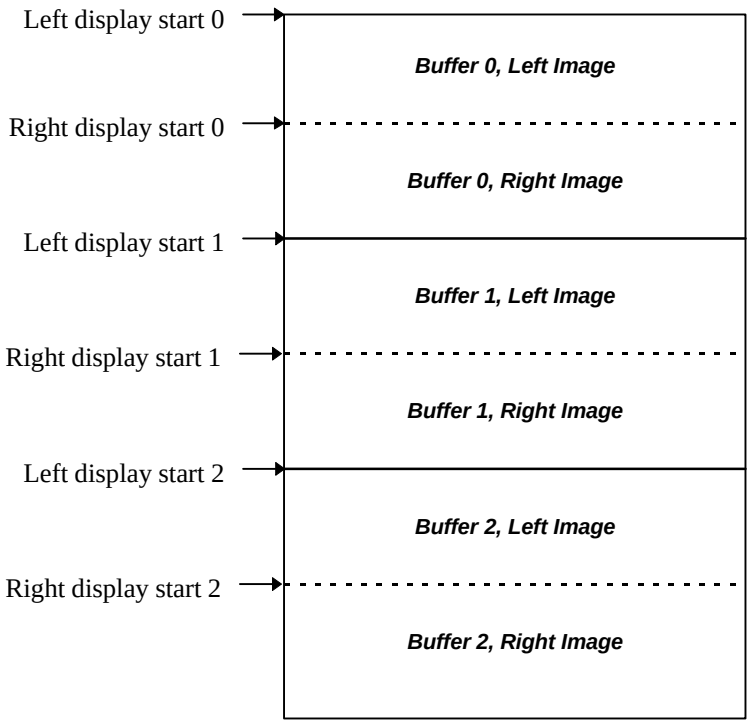


While the hardware has been programmed to display from buffer 0, it will continuously cycle between the left and right images in buffer 0. When the application request to change to buffer 1, it will change to continuously cycle between the left and right images for buffer 1. Note that if display start 1 is programmed while the hardware is displaying the left image in buffer 0, the hardware may switch to the right image in buffer 1 at the next vertical retrace or it may continue on and display the right image buffer 0, switching to buffer 1 after the buffer 0 right image has been displayed.

Automatic hardware display start address swapping (Method 2)

VBE 3.0 supports hardware implementations that can be programmed to automatically swap between the left and right images of the stereoscopic display automatically every vertical retrace. Normally when a mode is set, stereoscopic mode will be disabled and can be enabled by the application by calling function 4F07h, subfunctions 03h or 82h (and disabled with subfunction 06h). Once stereoscopic mode has been enabled, the hardware will display the left image from the data in video memory defined by the current left display start address, followed by the right image from the data in video memory defined by the current right display start address. After the right image has been displayed the hardware will begin displaying the left image again in a continuous cycle.

The diagram below shows how the pages could be laid out in video memory for a triple buffered, stereoscopic enabled graphics mode using a above/below approach:



While the hardware has been programmed to display from buffer 0, it will continuously cycle between the left and right images in buffer 0. When the application request to change to buffer 1, it will change to continuously cycle between the left and right images for buffer 1. Note that if display start 1 is programmed while the hardware is displaying the left image in buffer 0, the hardware may switch to the right image in buffer 1 at the next vertical retrace or it may continue on and display the right image buffer 0, switching to buffer 1 after the buffer 0 right image has been displayed.

Software driven display start address swapping

If the hardware does not support a free running stereoscopic display mode, the application must implement the free running display start address changes in software using a timer interrupt handler. The application should set the timer interrupt handler to run at a rate very close to the vertical refresh rate of the graphics mode being used, and use function 4F07h to swap the display start address during the vertical retrace interval.

Refresh rates and stereoscopic imaging

When the hardware is running in free running stereoscopic mode and an image or 3D scene is being viewed through LC shutter glasses, the user will see the resulting image at half the original refresh rate through the shutter glasses. Hence a normally acceptable display running at 60Hz becomes a hard to view display running at 30Hz stereoscopic. For this reason when running in stereoscopic modes it is desirable to significantly increase the refresh rate of the graphics mode to values as high as 120Hz or 140Hz (depending on the monitors capabilities), which provides for 60Hz or 70Hz refresh per eye in stereoscopic modes.

VBE 3.0 has full support for refresh rate control when setting a display mode via function 4F02h. Stereoscopic applications can use this functionality in combination with the GTF standard to increase the refresh rate of the stereoscopic application to acceptable levels (see the above section on refresh rate control for more information).

Left/right image synchronization

Signaling to the stereoscopic LC shutter glasses which image is currently being displayed is out of the scope of this specification, however the VBE/Core specification does include capabilities bits to let the application software know when a hardware stereoscopic synchronization signal is available. VESA is currently working on a new hardware standard for both the standard VGA and the new EVC connector's for signaling this synchronization information, and VBE function 4F00h will let the application know if stereoscopic signaling is available via the EVC connector or via the VESA stereoscopic signaling connector. Please contact VESA for more information on this new hardware signaling standard.

Developing for Maximum Compatibility

This section contains information relating to developing application software with maximum compatibility in mind, without sacrificing performance or features. Although the VBE standard defines how the specification should work, there are many different flavors of hardware out in the field. It is very important that you design your application with the following special cases in mind so that your application will run on the widest variety of hardware possible.

One of the common mistakes that many developers make when they first start developing graphics code with VBE devices is to assume the graphics card they have in their system is a representative sample of what exists in the field. Although the VBE interface will be identical on another graphics card, the capabilities and attributes that card reports may be different. This section also deals with explaining the most common pitfalls that plague developers first starting to develop VBE code.

This section is also very useful for the hardware vendor implementing a VBE BIOS for their graphics hardware, as it allows the hardware vendor to understand some of the more common pitfalls that developers will fall into, and to try and develop their BIOS code (and future hardware) to provide for the best compatibility in the field.

Be prepared for different Window Granularity's

One area that is of vital importance is the Windows Granularity, or the smallest increment that you can move the bank switching window. Many controllers simply provide a 64Kb granularity, which means that bank 0 starts at 0, bank 1 at 64Kb and bank 2 at 128Kb etc. However some controllers may provide either a 4Kb or a 16Kb granularity. For a 4Kb granularity controller, bank 0 starts at 0 while bank 1 starts at 4Kb not 64Kb. On a 4Kb system to move the window to the next 64Kb location you would need to program bank number 8 rather than bank number 1.

One method of adjusting the bank numbers in your bank switching code is outlined in the sample code in Appendix 2. This code basically finds a *shift factor* when the mode is initialized that indicates how many bits to shift the bank value left to adjust from a 64Kb window to the granularity of the hardware. In your bank switch code you simply shift the bank value left by the specified number of bits before calling the VBE bank switch function.

Be prepared for both single and dual read/write Windows

Another area of importance is if the controller has a single read/write window or separate read and write window's. On a controller with a single read/write window (the most common scenario), setting the first window will change the offset in the framebuffer where both reads and writes occur to the same location. On a controller with dual read/write window's, you can individually change the location in the framebuffer where read operations will occur and the location where write operations will occur.

To provide for maximum performance you should make sure you check the modeInfoBlock attributes for the graphics mode you are programming. If separate read/write windows are provided you must ensure that you set both windows to the same location with two calls to the bank switch function.

Be prepared to support both 15 and 16 bits per pixel high color modes

Many new games and applications have support for 15 and 16 bits per pixel high color modes. If you wish to support these modes, don't make the mistake of assuming that all devices will support 16 bit high color modes, or that all devices will support 15 bit high color modes. There are devices in the field that support only 15 bit modes, and there are also devices in the field that support only 16 bit modes. Hence it is of vital importance that your application code support both of these color depths to ensure maximum compatibility.

Be prepared to support both 24 and 32 bits per pixel true color modes

If you are developing code to support 24 bit true color rendering, be prepared to find controllers in the field that have the true color modes support as either 24 bits per pixel (3 bytes per pixel) or 32 bits per pixel (4 bytes per pixel). Generally the 32 bits per pixel modes are faster because the pixels can be written with a single CPU double word access, however 32 bits per pixel modes require a controller that has at least 2Mb of memory.

Hence with controllers that have 2Mb or more of memory, be prepared to find support for only 32 bits per pixel modes and no 24 bits modes.

Some controllers can't do double scanned modes

If you are developing a game or application that wishes to support 320x200 or 320x240 modes (in any color depth), be prepared for situations where these modes do not exist. To be able to initialize these modes on today's hardware requires support for double scanning, and there are some controllers in the field that do not support this. On these controllers these modes can never

be supported, so you application or game must be able to deal with the situation if these modes do not exist.

In lieu of these modes not being available, the controller may provide support for 320x400 and 320x480 modes which do not require double scanning (this is recommended for hardware vendors who don't have double scanning support). One neat solution is to support these modes by rendering your frames to a system memory buffer with a resolution 320x200 or 320x240, and then do a copy to the graphics screen with a 2x vertical stretch (just duplicate every scanline twice in software). The end result will look identical to a real 320x200 or 320x240 mode, and you will only lose a small amount in overall performance (and it will be much faster than rendering directly to a 320x400 or 320x480 screen for graphics intensive games).

Check if VGA Compatible Before Touching Any VGA Registers

Many developers find that there is an irresistible urge to push the boundaries of performance, and they will try anything and everything they can do attain these goals. One of the things that is commonly done is to perform weird and wonderful feats of magic using some of the standard VGA registers. This does work, and work well on VGA compatible graphics cards, but not on all cards!

If the graphics controller is based on a NonVGA graphics hardware technology (and many popular ones and many newer ones are), in the SuperVGA graphics modes the VGA registers simply do not exist anymore, and attempting to synch to these registers will put your code into an infinite loop. So be forewarned that doing any fiddling with the standard VGA registers is asking for trouble on certain graphics card that use NonVGA controllers to program the SuperVGA graphics modes (and lots more of these are coming out).

There is however a solution for VBE 2.0 and above controllers. There is a bit in the VBE modeInfoBlock for every graphics mode that indicates whether that mode is a NonVGA mode or VGA compatible mode. If this bit is set indicating that a NonVGA controller is being used to program the desired graphics mode, you must not do anything related to re-programming any of the standard VGA registers. In these cases you must fallback onto generic code that will perform all its graphics card interaction through the standard VBE 2.0 and above services.

Check if VGA Compatible Before Directly Programming the DAC

Another area of concern is programming the color palette in 256 color modes. Once again the same problem occurs when programming the palette for NonVGA controllers; the VGA palette registers no longer exist and attempting to program the palette via these registers will simply do nothing. Even worse attempting to synch to the vertical or horizontal retrace will also cause the system to get into an infinite loop.

Hence if you need to program the color palette on a NonVGA controller, you must use the supplied VBE 2.0 and above palette programming routines rather than programming the palette directly. Make sure you check the NonVGA attribute bit as discussed above, and if a NonVGA

mode is detected you will have to program the palette via the standard VBE 2.0 and above services.

VBE Function Reference

This chapter describes in detail each of the functions defined by the VBE standard. VBE functions are called from real mode using the INT 10h software interrupt vector, passing arguments in the 80X86 registers. The INT 10h interrupt handler first determines if a VBE function has been requested, and if so, processes that request. Otherwise control is passed to the standard VGA BIOS for completion. Starting with VBE 3.0, VBE functions can also be called directly from protected mode via the protected mode entry point. When called via the protected mode entry point, the VBE functions execute as 16-bit protected mode code which can be called directly from any 16-bit or 32-bit protected mode operating system or application.

All VBE functions are called with the AH register set to 4Fh to distinguish them from the standard VGA BIOS functions. The AL register is used to indicate which VBE function is to be performed. For supplemental or extended functionality the BL register is used when appropriate to indicate a specific subfunction.

Functions 00h-0Fh have been reserved for Standard VBE function numbers; Functions 10h-FFh are reserved for VBE Supplemental Specifications.

In addition to the INT 10h interface and protected mode entry point, a simple 32-bit Protected Mode Interface is available and is described below. In cases where there is both a standard interface and a 32-bit protected mode interface, the register parameters for both interfaces will be defined one after another for the function specification.

VBE Return Status

The AX register is used to indicate the completion status upon return from VBE functions (except for 32 bit protected mode functions; 32 bit versions of the functions do not return any status information or return codes). If VBE support for the specified function is available, the 4Fh value passed in the AH register on entry is returned in the AL register. If the VBE function completed successfully, 00h is returned in the AH register. Otherwise the AH register is set to indicate the nature of the failure.

VBE RETURN STATUS

AL == 4Fh:	Function is supported
AL != 4Fh:	Function is not supported
AH == 00h:	Function call successful
AH == 01h:	Function call failed
AH == 02h:	Function is not supported in the current hardware configuration
AH == 03h:	Function call invalid in current video mode

Note: Applications should treat any non-zero value in the AH register as a general failure condition as later versions of the VBE may define additional error codes.

VBE Mode Numbers

Standard VGA mode numbers are 7 bits wide and presently range from 00h to 13h. OEMs have defined extended display modes in the range 14h to 7Fh. Values from 80h to FFh cannot be used, since VGA BIOS Function 00h (Set video mode) interprets bit 7 as a flag to clear or preserve display memory.

Due to the limitations of 7-bit mode numbers, the optional VBE mode numbers are 14 bits wide. To initialize a VBE mode, the mode number is passed in the BX register to VBE Function 02h (Set VBE mode).

The format of VBE mode numbers is as follows:

D0-D8	=	Mode number If D8 == 0, this is not a VESA defined VBE mode If D8 == 1, this is a VESA defined VBE mode
D9-D12	=	Reserved by VESA for future expansion (= 0)
D11	=	Refresh Rate Control Select If D11 == 0, Use current BIOS default refresh rate If D11 == 1, Use user specified CRTC values for refresh rate
D12-13	=	Reserved for VBE/AF (must be 0)
D14	=	Linear/Flat Frame Buffer Select If D14 == 0, Use Banked/Windowed Frame Buffer If D14 == 1, Use Linear/Flat Frame Buffer
D15	=	Preserve Display Memory Select If D15 == 0, Clear display memory If D15 == 1, Preserve display memory

Thus, VBE mode numbers begin at 100h. This mode numbering scheme implements standard 7-bit mode numbers for OEM-defined modes (OEM defined modes are those that can be set via the Standard VGA BIOS). Standard VGA modes may be initialized through VBE Function 02h (Set VBE mode) simply by placing the mode number in BL and clearing the upper byte (BH). 7-bit OEM-defined display modes may be initialized in the same way. Note that VBE modes may only be set if the mode exists in the VideoModeList pointed to by the VideoModePTR returned in Function 00h, while Standard VGA modes and OEM defined 7 bit modes may be initialized without a corresponding entry in the VideoModeList and a mode info block. The exception to this requirement is the VBE mode number 81FFh.

To date, VESA has defined one special 7-bit mode number, 6Ah, for the 800x600, 16-color, 4-plane graphics mode. The corresponding 15-bit mode number for this mode is 102h.
The following VBE mode numbers have been previously defined by VESA:

GRAPHICS				TEXT			
15-bit mode number	7-bit mode number	Resolution	Colors	15-bit mode number	7-bit mode number	Columns	Rows
100h	-	640x400	256	108h	-	80	60
101h	-	640x480	256	109h	-	132	25
102h	6Ah	800x600	16	10Ah	-	132	43
103h	-	800x600	256	10Bh	-	132	50
104h	-	1024x768	16	10Ch	-	132	60
105h	-	1024x768	256				
106h	-	1280x1024	16				
107h	-	1280x1024	256				
10Dh	-	320x200	32K (1:5:5:5)				
10Eh	-	320x200	64K (5:6:5)				
10Fh	-	320x200	16.8M (8:8:8)				
110h	-	640x480	32K (1:5:5:5)				
111h	-	640x480	64K (5:6:5)				
112h	-	640x480	16.8M (8:8:8)				
113h	-	800x600	32K (1:5:5:5)				
114h	-	800x600	64K (5:6:5)				
115h	-	800x600	16.8M (8:8:8)				
116h	-	1024x768	32K (1:5:5:5)				
117h	-	1024x768	64K (5:6:5)				
118h	-	1024x768	16.8M (8:8:8)				
119h	-	1280x1024	32K (1:5:5:5)				
11Ah	-	1280x1024	64K (5:6:5)				
11Bh	-	1280x1024	16.8M (8:8:8)				
81FFh		Special Mode (see below for details)					

Note: Starting with VBE version 2.0, VESA will no longer define new VESA mode numbers and it will no longer be mandatory to support these old mode numbers. OEM's who wish to add new VBE mode numbers to their implementations, must make sure that all new modes are defined with mode numbers above 0x100, However, it is highly recommended that BIOS implementations continue to support these mode numbers for compatibility with older software and add new mode numbers after the last VESA defined mode number). VBE 2.0-aware applications should follow the guidelines in Appendix 5 - Application Programming Considerations - for setting a desired mode.

Note: Mode 81FFh is a special mode designed to preserve the current memory contents and give access to the entire video memory. This mode is especially useful for saving the entire video memory contents before going into a state that could lose the contents (e.g., set this mode to gain access to all video memory to save it before going into a volatile power down state). This mode is required because the entire video memory contents are not always accessible in every mode. It is recommended that this mode be packed pixel in format, and a ModeInfoBlock must be defined for it. However, it should not appear in the

VideoModeList. Look in the ModeInfoBlock to determine if paging is required and how paging is supported if it is. Also note that there are no implied resolutions or timings associated with this mode.

Note: Future display resolutions will be defined by VESA display vendors. The color depths will not be specified and new mode numbers will *not* be assigned for these resolutions. For example, if the VESA display vendors define 1600x1200 as a VESA resolution, application developers should target their display resolution for 1600x1200 rather than choosing an arbitrary resolution like 1550x1190. The VBE implementation should be queried to get the available resolutions and color depths and the application should be flexible enough to work with this list. Appendix 5 gives a detailed summary of the way an application should go about selecting and setting modes.

VBE Far Pointers

Throughout this specification references will be made to a pointer of the type 'vbeFarPtr'. This is a DWORD pointer that can have two different interpretations depending on whether the BIOS is being called via the real mode INT 10h software interrupt, or via the protected mode entry point. When functions are called via the real mode INT 10h software interrupt, a 'vbeFarPtr' will be a real mode segment:offset style pointer to a memory location below the 1Mb system memory boundary. When functions are called via the protected mode entry point, a 'vbeFarPtr' will be a valid 16-bit protected mode selector:offset style pointer, the selector of which may point to the base of the protected mode BIOS image loaded in memory, user data passed in to the protected mode BIOS or to the information blocks passed in to the protected mode BIOS. In any case the calling application and BIOS can simply reference the pointer as a 32-bit far pointer to access the data, but should avoid doing any real mode specific pointer arithmetic on the selector:offset values.

Obtaining the Protected Mode Entry Point

Starting with VBE/Core 3.0, all the VBE functions are optionally accessible from 16-bit and 32-bit protected mode applications and operating systems via a new 'Protected Mode Entry Point'. The protected mode entry point defines a special location that can be used to directly call the VBE functions as 16-bit protected mode code. The application or OS does not call the BIOS code directly from protected mode, but first makes a copy of the BIOS image in a writeable section of memory and then calls the code within this relocated memory block. The entry point is located within a special 'Protected Mode Information Block', which will be located somewhere within the first 32Kb of the BIOS image (if this information block is not found, then the BIOS does not support this new interface). The PM Info Block structure is defined as follows:

```
PMInfoBlock struc
Signature       db      'PMID'      ; PM Info Block Signature
EntryPoint      dw      ?           ; Offset of PM entry point within BIOS
PMInitialize    dw      ?           ; Offset of PM initialization entry point
BIOSDataSel     dw      0           ; Selector to BIOS data area emulation block
A0000Sel        dw      A000h       ; Selector to access A0000h physical mem
B0000Sel        dw      B000h       ; Selector to access B0000h physical mem
B8000Sel        dw      B800h       ; Selector to access B8000h physical mem
CodeSegSel      dw      C000h       ; Selector to access code segment as data
InProtectMode   db      0           ; Set to 1 when in protected mode
Checksum        db      ?           ; Checksum byte for structure
PMInfoBlock ends
```

To find the PMInfoBlock the application must scan the BIOS image looking for the appropriate signature. When a potential info block has been found, the application must then perform a checksum on the PMInfoBlock by adding the byte values of all entries in the block. If the block is valid, the sum of all bytes should be zero (the Checksum byte is used to force the result of the sum to zero).

Note: The protected mode entry point is optional, and may not be implemented in some VBE 3.0 BIOS'es. Also note that the VESA VBE/AF Accelerator Functions specifications provide an alternative protected mode environment, so application programmers looking for protected mode support may want to take a look at the VBE/AF specification.

Description of the **PMInfoBlock** structure fields:

The **Signature** field contains the special signature that is used to identify the PMInfoBlock located within the BIOS image.

The **EntryPoint** field contains the offset of the Protected Mode Entry Point from the start of the BIOS image. This offset is used to create a pointer for directly calling the protected mode entry point from both 16-bit and 32-bit protected mode code.

The **PMInitialize** field contains the offset of a function for initializing the protected mode BIOS code. The calling application must fill in the PMInfo block with valid selectors for protected mode operation, and then call the PMInitialize function before calling any functions via the protected mode entry point. The PMInitialize function is responsible for initializing any internal variables in the BIOS that may be different in protected mode such as internal pointers to font tables or other data (using the CodeSegSel selector to address variables within the code segment).

The **BIOSDataSel** field contains a protected mode selector that references a memory block of at least 600h bytes in length that will be cleared to all zeroes. This data block is used to provide a memory region that the BIOS can use for caching information across calls to the BIOS functions, and essentially will be used to emulate the first 600h bytes of low DOS memory which will no longer be present when running in protected mode. The real mode BIOS will always set this field to 0 so that real mode code can directly access the real BIOS data area. If the BIOS relies on certain values being present in this data block, the PMInitialize function should be used to fill in this block with default values for protected mode operation. This selector should be a 16-bit data selector with a minimum limit of 600h bytes with read/write permissions.

The **A0000Sel**, **B0000Sel** and **B8000Sel** field's contain selectors that point to the A0000h, B0000h and B8000h physical memory areas. These selectors are used by the protected mode BIOS code when it needs to access these physical memory areas (for clearing the screen, or accessing memory mapped registers etc.). The real mode BIOS will always set these fields to the real mode segment values initially so that real mode BIOS code can directly access these memory areas. These selectors should be 16-bit data selectors with a limit of 64Kb with read/write permissions (32Kb for B800Sel).

The **CodeSegSel** field contains a protected mode data selector that provides read/write access to the BIOS image loaded in memory. This selector can be used by the protected mode code during the PMInitialize function to self-modify portions of the code or internal pointers used during protected mode operation. The real mode BIOS will always set this field to a value of C000h so that real mode BIOS code can directly access the code segment. This selector should be a 16-bit data selector with a limit of 64Kb with read/write permissions (it is not executable).

The **InProtectMode** field is used to indicate to the BIOS code that it is running in protected mode. By default the real mode BIOS will set this value to 0 to indicate that it is running in real mode, and when the BIOS image has been relocated by the application, it will set this field to a '1'. The BIOS code can then use this field to determine if it is running in real mode or protected mode to fail functions that are not supported by the protected mode entry point.

So to summarize, the steps involved in finding the protected mode entry point and setting up to call the protected mode BIOS code is as follows:

1. Allocate a protected mode buffer large enough to hold the entire BIOS image (32Kb normally).

2. Copy the BIOS image from the C0000h physical memory region to the protected mode buffer.

3. Scan the BIOS image for the PMInfoBlock structure and check that the checksum for the block is 0.

4. Allocate an empty (all zeros) block of memory for the BIOS data area emulation block that is at least 600h bytes in length. Create a new data selector that points to the start of this memory block and put the value of that selector into the BIOSDataSel field of the PMInfoBlock.

5. Create selectors that point to the A0000h, B0000h and B8000h physical memory locations (with lengths of 64Kb, 64Kb and 32Kb respectively). Put the values of these selectors into the A0000Sel, B0000Sel and B8000Sel fields of the PMInfoBlock respectively.

6. Create a read/write data selector that points to the loaded BIOS image and put the value into the CodeSegSel field of the PMInfoBlock.

7. Set the InProtectMode field of the PMInfoBlock to a '1' to indicate to the BIOS code that it is now running in protected mode.

8. Create a 16-bit code segment selector (execute and read permissions) that points to the start of the protected mode BIOS buffer you have allocated. This selector and the PMInitialize entry point from the PMInfoBlock provide a 16:16 or 16:32 (the offset must be extended to 32-bit under a 32-bit OS) far pointer to the protected mode BIOS initialization function. You should call this function first before calling any of the functions via the protected mode entry point.

9. Create a 16-bit data segment selector that points to a region of memory at least 1024 bytes in size that will be used as the protected mode stack for calls to the protected mode BIOS code. This selector will have to be loaded into the SS selector before the call, and the SP register should be cleared to zero (i.e.: SS:0 points to the start of the 16-bit protected mode stack). The 16-bit protected mode stack is necessary so that if the BIOS is using the stack to maintain local variables at runtime, those variables can be correctly referenced via the 16-bit stack pointer.

10. Using the selector created in the above step and the protected mode entry point from the PMInfoBlock, you now have a 16:16 or 16:32 (the offset must be extended to 32-bit under a 32-bit OS) far pointer to the protected mode BIOS, which you can simply call directly. Note that you do need to ensure that the stack is switched to the 16-bit protected mode stack before the protected mode BIOS is called.

Calling the Protected Mode Entry Point

Calling the protected mode entry point is almost identical to calling the real mode INT 10h functions. The only difference is that instead of issuing an INT 10h software interrupt, your code will simply make a far call via the 16:16 or 16:32 pointer that defines the entry point for the protected mode BIOS. A far call from 32-bit protected mode (16:32 pointer) to the protected

mode entry point will put the CPU into 16-bit protected mode, since the BIOS code segment is a 16-bit selector.

Note however that any values passed in to the protected mode BIOS such as memory buffers passed in ES:DI, will need to be valid 16:16 protected mode values. For 32-bit environments you cannot simply pass in the 32-bit near pointers to the protected mode BIOS code, but must translate them to 16:16 protected mode pointers. The steps to do this are relatively simple as outlined below:

1. Find the linear base address of the default DS selector for your 32-bit data.
2. Add the above linear address to the 32-bit offset of the buffer you need to pass.
3. Create a new data selector that points to the linear base address calculated in step 2 above, with a limit large enough to cover the entire buffer you are passing.
4. Call the VBE function with ES set to the selector created in step 3 and an offset of 0.
5. Destroy the selector created in step 3 (or alternatively re-use the same selector, just re-assign the base address and length for every function call).

Protected Mode Entry Point Functional Restrictions

The protected mode entry point provides the capability to call and use the VBE defined functions from 16-bit and 32-bit protected mode. However the following restrictions may apply (BIOS vendors may implement the functionality if they feel so inclined, but it is not mandatory):

- No support for any of the Standard VGA BIOS functions, only the VBE interface functions. Calling the protected mode entry point for non-VBE functions may produce unpredictable results.
- No support for extended text modes, only graphics modes. Extended text modes require access to VGA BIOS functionality which may not be provided via the protected mode entry point.
- None of the ‘Get’ style functions such as 4F03h (Get Current Video Mode) or 4F07h (Get Display Start) that return status information are supported by the protected mode entry point. It is assumed this information will be cached by the operating system or application internally. Calling these functions via the protected mode entry point may result in undefined behavior.
- Note that supplemental VBE services are not be restricted and BIOS implementers are free to implement these services via the protected mode entry point.

Function 00h - Return VBE Controller Information

Input:	AX	= 4F00h	Return VBE Controller Information
	ES:DI	=	Pointer to buffer in which to place VbeInfoBlock structure (VbeSignature should be set to 'VBE2' when function is called to indicate VBE 3.0 information is desired and the information block is 512 bytes in size.)
Output:	AX	=	VBE Return Status

Note: All other registers are preserved.

This required function returns the capabilities of the display controller, the revision level of the VBE implementation, and vendor specific information to assist in supporting all display controllers in the field.

The purpose of this function is to provide information to the calling program about the general capabilities of the installed VBE software and hardware. This function fills an information block structure at the address specified by the caller. The VbeInfoBlock information block size is 256 bytes for VBE 1.x, and 512 bytes for VBE 2.0+.

The information block has the following structure:

```
VbeInfoBlock struc
VbeSignature        db      'VESA'          ; VBE Signature
VbeVersion          dw      0300h           ; VBE Version
OemStringPtr        dd      ?               ; VbeFarPtr to OEM String
Capabilities        db      4 dup (?)       ; Capabilities of graphics controller
VideoModePtr        dd      ?               ; VbeFarPtr to VideoModeList
TotalMemory         dw      ?               ; Number of 64kb memory blocks
                                            ; Added for VBE 2.0+
OemSoftwareRev      dw      ?               ; VBE implementation Software revision
OemVendorNamePtr    dd      ?               ; VbeFarPtr to Vendor Name String
OemProductNamePtr   dd      ?               ; VbeFarPtr to Product Name String
OemProductRevPtr    dd      ?               ; VbeFarPtr to Product Revision String
Reserved            db      222 dup (?)     ; Reserved for VBE implementation scratch
                                            ;   area
OemData             db      256 dup (?)     ; Data Area for OEM Strings
VbeInfoBlock ends
```

Note:	All data in this structure is subject to change by the VBE implementation when VBE Function 00h is called. Therefore, it should not be used by the application to store data of any kind.

Description of the **VbeInfoBlock** structure fields:

The **VbeSignature** field is filled with the ASCII characters 'VESA' by the VBE implementation. VBE 2.0+ applications should preset this field with the ASCII characters 'VBE2' to indicate to the VBE implementation that the VBE 2.0 extended information is desired, and the VbeInfoBlock is 512 bytes in size. Upon return from VBE Function 00h, this field should always be set to 'VESA' by the VBE implementation.

The **VbeVersion** is a BCD value which specifies what level of the VBE standard is implemented in the software. The higher byte specifies the major version number. The lower byte specifies the minor version number.

Note:	The BCD value for VBE 3.0 is 0300h and the BCD value for VBE 1.2 is 0102h. In the past we have had some applications misinterpreting these BCD values. For example, BCD 0102h was interpreted as 1.02, which is incorrect.

The **OemStringPtr** is a VbeFarPtr to a null terminated OEM-defined string. This string may be used to identify the graphics controller chip or OEM product family for hardware specific display drivers. There are no restrictions on the format of the string. This pointer may point into either ROM or RAM, depending on the specific implementation. VBE 3.0 BIOS implementations may place this string in the OemData area within the VbeInfoBlock if 'VBE2' is preset in the VbeSignature field on entry to Function 00h.

Note:	The length of the OEMString is not defined, but for space considerations, we recommend a string length of less than 256 bytes.

The **Capabilities** field indicates the support of specific features in the graphics environment. The bits are defined as follows:

D0	= 0	DAC is fixed width, with 6 bits per primary color
	= 1	DAC width is switchable to 8 bits per primary color
D1	= 0	Controller is VGA compatible
	= 1	Controller is not VGA compatible
D2	= 0	Normal RAMDAC operation
	= 1	When programming large blocks of information to the RAMDAC, use the blank bit in Function 09h.
D3	= 0	No hardware stereoscopic signaling support
	= 1	Hardware stereoscopic signaling supported by controller
D4	= 0	Stereo signaling supported via external VESA stereo connector
	= 1	Stereo signaling supported via VESA EVC connector
D5-31	= Reserved	

BIOS Implementation Note: The DAC must always be restored to 6 bits per primary as default upon a mode set. If the DAC has been switched to 8 bits per primary, the mode set must restore the DAC to 6 bits per primary to so the application developer does not have to reset it.

Application Developer's Note: If a DAC is switchable, you can assume that the DAC will be restored to 6 bits per primary upon a mode set. For an application to use a DAC, the application program is responsible for setting the DAC to 8 bits per primary mode using Function 08h.

VGA compatibility is defined as supporting all standard IBM VGA modes, fonts and I/O ports; however, VGA compatibility doesn't guarantee that all modes which can be set are VGA compatible, or that the 8x14 font is available.

The need for D2 = 1 "program the RAMDAC using the blank bit in Function 09h" is for older style RAMDACs, where programming the RAM values during display time causes a "snow-like" effect on the screen. Some newer style RAMDACs don't have this limitation and can easily be programmed at any time, but older RAMDACs require that they be programmed during the vertical retrace period so as not to display the snow while values change during display time. This bit informs the software that it should make the function call with BL=80h rather than BL=00h to ensure the minimization of the "snow-like" effect.

Bit D3 will be set to a 1 if the hardware supports the new VESA stereoscopic synchronization signal either via the VESA stereo connector or via the VESA EVC connector. This bit is primarily intended for informational purposes, so that application software can automatically use hardware signaling when it is available, and default back to a software signaling method when one is not available.

Bit D4 will be set to a 1 if a VESA EVC connector is available and the stereoscopic synchronization signal is available on this connector. Bit D4 will be set to a 0 if the VESA stereo connector contains the synchronization information. If bit D3 is 0, this bit will be set to 0 by the BIOS, and should be ignored by the application programmer. Note that if a VESA EVC connector is present in the system, it is mandatory that it be used for the stereoscopic synchronization signal.

The **VideoModePtr** is a VbeFarPtr that points to a list of mode numbers for all display modes supported by the VBE implementation. Each mode number occupies one word (16 bits). The list of mode numbers is terminated by a -1 (0FFFFh). The mode numbers in this list represent all of the potentially supported modes by the display controller. Refer to Chapter 3 for a description of VESA VBE mode numbers. VBE 3.0 BIOS implementations may place this mode list in the Reserved area of the VbeInfoBlock if 'VBE2' is preset in the VbeSignature field on entry to Function 00h, or have it statically stored within the VBE implementation.

Note:	It is responsibility of the application to verify the actual availability of any mode returned by this function through the Return VBE Mode Information (VBE Function 01h) call. Some of the returned modes may not be available due to the actual amount of memory physically installed on the display board or due to the capabilities of the attached monitor.

Note: If a VideoModeList is found to contain no entries (starts with 0FFFFh), it can be assumed that the VBE implementation is a "stub" implementation where only Function 00h is supported for diagnostic or "Plug and Play" reasons. These stub implementations are not VBE 3.0 compliant and should only be implemented in cases where no space is available to implement the entire VBE.

Note: The VBE 3.0 protected mode entry point may not provide support for extended text modes nor Standard VGA graphics modes. Any attempt to set one of these modes via the protected mode entry point will have an undefined effect.

The **TotalMemory** field indicates the maximum amount of memory physically installed and available to the frame buffer in 64KB units. (e.g. 256KB = 4, 512KB = 8) Not all video modes can address all this memory, see the ModeInfoBlock for detailed information about the addressable memory for a given mode.

The **OemSoftwareRev** field is a BCD value which specifies the OEM revision level of the VBE software. The higher byte specifies the major version number. The lower byte specifies the minor version number. This field can be used to identify the OEM's VBE software release. This field is only filled in when 'VBE2' is preset in the VbeSignature field on entry to Function 00h.

The **OemVendorNamePtr** is a VbeFarPtr to a null-terminated string containing the name of the vendor which produced the display controller board product. (This string may be contained in the VbeInfoBlock or the VBE implementation.) This field is only filled in when 'VBE2' is preset in the VbeSignature field on entry to Function 00h. (**Note:** the length of the strings OemProductRev, OemProductName and OemVendorName (including terminators) summed, must fit within a 256 byte buffer; this is to allow for return in the OemData field if necessary.).

The **OemProductNamePtr** is a VbeFarPtr to a null-terminated string containing the product name of the display controller board. (This string may be contained in the VbeInfoBlock or the VBE implementation.) This field is only filled in when 'VBE2' is preset in the VbeSignature field on entry to Function 00h. (**Note:** the length of the strings OemProductRev, OemProductName and OemVendorName (including terminators) summed, must fit within a 256 byte buffer; this is to allow for return in the OemData field if necessary.).

The **OemProductRevPtr** is a VbeFarPtr to a null-terminated string containing the revision or manufacturing level of the display controller board product. (This string may be contained in the VbeInfoBlock or the VBE implementation.) This field can be used to determine which production revision of the display controller board is installed. This field is only filled in when 'VBE2' is preset in the VbeSignature field on entry to Function 00h. (**Note:** the length of the strings OemProductRev, OemProductName and OemVendorName (including terminators) summed, must fit within a 256 byte buffer; this is to allow for return in the OemData field if necessary.).

The **Reserved** field is a space reserved for dynamically building the VideoModeList if necessary if the VideoModeList is not statically stored within the VBE implementation. This field should not

be used for anything else, and may be reassigned in the future. Application software should not assume that information in this field is valid.

The **OemData** field is a 256 byte data area that is used to return OEM information returned by VBE Function 00h when 'VBE2' is preset in the VbeSignature field. The OemVendorName string, OemProductName string and OemProductRev string may be copied into this area by the VBE implementation. This area will only be used by VBE implementations 2.0 and above when 'VBE2' is preset in the VbeSignature field.

Function 01h - Return VBE Mode Information

Input:	AX	= 4F01h	Return VBE mode information
	CX	=	Mode number
	ES:DI	=	Pointer to ModeInfoBlock structure
Output:	AX	=	VBE Return Status

Note: All other registers are preserved.

This required function returns extended information about a specific VBE display mode from the mode list returned by VBE Function 00h. This function fills the mode information block, ModeInfoBlock, structure with technical details on the requested mode. The ModeInfoBlock structure is provided by the application with a fixed size of 256 bytes.

Information can be obtained for all listed modes in the VideoModeList returned in Function 00h. If the requested mode cannot be used or is unavailable, a bit will be set in the ModeAttributes field to indicate that the mode is not supported in the current configuration.

The mode information block has the following structure:

```
ModeInfoBlock struc

; Mandatory information for all VBE revisions
ModeAttributes      dw   ?    ; mode attributes
WinAAttributes      db   ?    ; window A attributes
WinBAttributes      db   ?    ; window B attributes
WinGranularity      dw   ?    ; window granularity
WinSize             dw   ?    ; window size
WinASegment         dw   ?    ; window A start segment
WinBSegment         dw   ?    ; window B start segment
WinFuncPtr          dd   ?    ; real mode pointer to window function
BytesPerScanLine    dw   ?    ; bytes per scan line

; Mandatory information for VBE 1.2 and above
XResolution         dw   ?    ; horizontal resolution in pixels or characters[3]
YResolution         dw   ?    ; vertical resolution in pixels or characters
XCharSize           db   ?    ; character cell width in pixels
YCharSize           db   ?    ; character cell height in pixels
NumberOfPlanes      db   ?    ; number of memory planes
BitsPerPixel        db   ?    ; bits per pixel
NumberOfBanks       db   ?    ; number of banks
```

[3]Pixels in graphics modes, characters in text modes.

```
MemoryModel         db      ?       ; memory model type
BankSize                    db      ?       ; bank size in KB
NumberOfImagePages          db      ?       ; number of images
Reserved                    db      1       ; reserved for page function

; Direct Color fields (required for direct/6 and YUV/7 memory models)
RedMaskSize                 db      ?       ; size of direct color red mask in bits
RedFieldPosition            db      ?       ; bit position of lsb of red mask
GreenMaskSize               db      ?       ; size of direct color green mask in bits
GreenFieldPosition          db      ?       ; bit position of lsb of green mask
BlueMaskSize                db      ?       ; size of direct color blue mask in bits
BlueFieldPosition           db      ?       ; bit position of lsb of blue mask
RsvdMaskSize        db      ?       ; size of direct color reserved mask in bits
RsvdFieldPosition           db      ?       ; bit position of lsb of reserved mask
DirectColorModeInfo         db      ?       ; direct color mode attributes

; Mandatory information for VBE 2.0 and above
PhysBasePtr                 dd      ?       ; physical address for flat memory frame buffer
Reserved                    dd      0       ; Reserved - always set to 0
Reserved                    dw      0       ; Reserved - always set to 0

; Mandatory information for VBE 3.0 and above
LinBytesPerScanLine         dw      ?       ; bytes per scan line for linear modes
BnkNumberOfImagePages       db      ?       ; number of images for banked modes
LinNumberOfImagePages       db      ?       ; number of images for linear modes
LinRedMaskSize              db      ?       ; size of direct color red mask (linear modes)
LinRedFieldPosition         db      ?       ; bit position of lsb of red mask (linear modes)
LinGreenMaskSize            db      ?       ; size of direct color green mask  (linear modes)
LinGreenFieldPositiondb     ?       ; bit position of lsb of green mask (linear modes)
LinBlueMaskSize             db      ?       ; size of direct color blue mask  (linear modes)
LinBlueFieldPosition        db      ?       ; bit position of lsb of blue mask (linear modes)
LinRsvdMaskSize             db      ?       ; size of direct color reserved mask (linear modes)
LinRsvdFieldPosition        db      ?       ; bit position of lsb of reserved mask (linear modes)
MaxPixelClock               dd      ?       ; maximum pixel clock (in Hz) for graphics mode

Reserved                    db      189 dup (?)  ; remainder of ModeInfoBlock
ModeInfoBlock ends
```

The **ModeAttributes** field describes certain important characteristics of the graphics mode.

The ModeAttributes field is defined as follows:

D0	=	Mode supported by hardware configuration
	0 =	Mode not supported in hardware
	1 =	Mode supported in hardware
D1	=	1 (Reserved)
D2	=	TTY Output functions supported by BIOS
	0 =	TTY Output functions not supported by BIOS
	1 =	TTY Output functions supported by BIOS
D3	=	Monochrome/color mode (see note below)
	0 =	Monochrome mode
	1 =	Color mode
D4	=	Mode type
	0 =	Text mode
	1 =	Graphics mode
D5	=	VGA compatible mode
	0 =	Yes
	1 =	No
D6	=	VGA compatible windowed memory mode is available
	0 =	Yes
	1 =	No
D7	=	Linear frame buffer mode is available
	0 =	No
	1 =	Yes
D8	=	Double scan mode is available
	0 =	No
	1 =	Yes
D9	=	Interlaced mode is available
	0 =	No
	1 =	Yes
D10	=	Hardware triple buffering support
	0 =	No
	1 =	Yes
D11	=	Hardware stereoscopic display support
	0 =	No
	1 =	Yes
D12	=	Dual display start address support
	0 =	No
	1 =	Yes
D13-D15	=	Reserved

Bit D0 is set to indicate that this mode can be initialized in the present hardware configuration. This bit is reset to indicate the unavailability of a graphics mode if it requires a certain monitor type, more memory than is physically installed, etc.

Bit D1 was used by VBE 1.0 and 1.1 to indicate that the optional information following the BytesPerScanLine field were present in the data structure. This information became mandatory with VBE version 1.2 and above, so D1 is no longer used and should be set to 1. The Direct Color fields are valid only if the MemoryModel field is set to a 6 (Direct Color) or 7 (YUV).

Bit D2 indicates whether the video BIOS has support for output functions like TTY output, scroll, etc. in this mode. TTY support is recommended but not required for all extended text and graphic modes. If bit D2 is set to 1, then the INT 10h BIOS must support all of the standard output functions listed below.

All of the following TTY functions must be supported when this bit is set:

01	Set Cursor Size
02	Set Cursor Position
06	Scroll TTY window up or Blank Window
07	Scroll TTY window down or Blank Window
09	Write character and attribute at cursor position
0A	Write character only at cursor position
0E	Write character and advance cursor

Bit D3 is set to indicate color modes, and cleared for monochrome modes.

Bit D4 is set to indicate graphics modes, and cleared for text modes.

Note: Monochrome modes map their CRTC address at 3B4h. Color modes map their CRTC address at 3D4h. Monochrome modes have attributes in which only bit 3 (video) and bit 4 (intensity) of the attribute controller output are significant. Therefore, monochrome text modes have attributes of off, video, high intensity, blink, etc. Monochrome graphics modes are two plane graphics modes and have attributes of off, video, high intensity, and blink. Extended two color modes that have their CRTC address at 3D4h, are color modes with one bit per pixel and one plane. The standard VGA modes, 06h and 11h, would be classified as color modes, while the standard VGA modes 07h and 0Fh would be classified as monochrome modes.

Bit D5 is used to indicate if the mode is compatible with the VGA hardware registers and I/O ports. If this bit is set, then the mode is NOT VGA compatible and no assumptions should be made about the availability of any VGA registers. If clear, then the standard VGA I/O ports and frame buffer address defined in WinASegment and/or WinBSegment can be assumed.

Bit D6 is used to indicate if the mode provides Windowing or Banking of the frame buffer into the frame buffer memory region specified by WinASegment and WinBSegment. If set, then Windowing of the frame buffer is NOT possible. If clear, then the device is capable of mapping the frame buffer into the segment specified in WinASegment and/or WinBSegment. (This bit is used in conjunction with bit D7, see table following D7 for usage).

Bit D7 indicates the presence of a Linear Frame Buffer memory model. If this bit is set, the display controller can be put into a flat memory model by setting the mode (VBE Function 02h) with the Flat Memory Model bit set. (This bit is used in conjunction with bit D6, see following table for usage)

	D7	D6
Windowed frame buffer only	0	0
n/a	0	1
Both Windowed and Linear[4]	1	0
Linear frame buffer only	1	1

Bit D8 indicates if the video mode can support double scanning or not. If this bit is set, the video mode can be initialized with the double scan flag set and the vertical resolution of the mode will be half the value of the vertical CRTC values. Double scanning is necessary to be able to support 200, 240 and 300 scanline graphics modes on modern controllers. Note that all 200, 240 and 300 scanline modes will have the double scan bit set.

Bit D9 indicates if the video mode can support interlaced operation or not. If this bit is set, the video mode can be initialized with the interlaced flag set and the controller will be initialized for an interlaced graphics mode. Note that some controllers may not support interlaced operation, so you must check this bit before attempting to initialize an interlaced mode.

Bit D10 indicates if the video mode can support hardware triple buffering or not. If this bit is set, the application program can use Function 4F07h, subfunction 04h to implement hardware triple buffering. If hardware triple buffering is not supported, the application program may use the new 02/82h subfunctions to set the display start address, but cannot use subfunction 04h to get status information on the scheduled display start address change.

Bit D11 indicates if the video mode can support hardware stereoscopic displays or not (for LC shutter glasses). If this bit is set, the application program can use Function 4F07h, subfunctions 02h/05/06h/82h to implement hardware stereoscopic display buffering. If bit D12 is also set, applications can take advantage of dual display start address hardware when present. If bit D10 is also set, applications can use subfunction 04h to get status information on the scheduled display start address change. Note that it is possible for hardware to support stereoscopic display buffering but not support hardware triple buffering.

Bit D12 indicates if the video mode can support dual display start addresses or not (for LC shutter glasses). If this bit is set, the application program can use Function 4F07h, subfunctions 03h/83h to implement hardware stereoscopic display buffering using the dual display start address capabilities, allowing the application to directly program the locations of the left and right image

[4]Use D14 of the Mode Number to select the Linear Buffer on a mode set (Function 02h).

buffers. If this bit is not set, applications will have to ensure that the left and right images are consecutive in memory as explained in the section on using hardware stereoscopic above.

The **BytesPerScanLine** field specifies how many full bytes are in each logical scanline in banked modes. The logical scanline could be equal to or larger than the displayed scanline (the number of physical pixels displayed). For linear modes please refer to the LinBytesPerScanLine field.

The **WinAAttributes** and **WinBAttributes** describe the characteristics of the CPU windowing scheme such as whether the windows exist and are read/writeable, as follows:

D0	=	Relocatable window(s) supported
	0 =	Single non-relocatable window only
	1 =	Relocatable window(s) are supported
D1	=	Window readable
	0 =	Window is not readable
	1 =	Window is readable
D2	=	Window writeable
	0 =	Window is not writeable
	1 =	Window is writeable
D3-D7	=	Reserved

Even if windowing is not supported (bit D0 = 0 for both Window A and Window B), then an application can assume that the display memory buffer resides at the location specified by WinASegment and/or WinBSegment.

WinGranularity specifies the smallest boundary, in KB, on which the window can be placed in the frame buffer memory. The value of this field is undefined if Bit D0 of the appropriate WinAttributes field is not set.

WinSize specifies the size of the window in KB.

Note: Don't confuse the WinGranularity field with the WinSize field. Most if not all controllers have a WinSize of 64Kb and it is a safe assumption that this will always be the case. The WinGranularity field may however be a value other than 64Kb, and some VBE BIOS'es may have it set to 4Kb or 16Kb.

WinASegment and **WinBSegment** address specify the segment addresses where the windows are located in the CPU address space. Note that these values are real mode segment values, so to convert the real 32 bit physical address you need to shift the values left 4 bits (ie: segment A000h is physical address A0000h). Also note that if the hardware has only linear framebuffer modes available, the values listed in here will be set to 0 indicating the banked framebuffer is not available.

WinFuncPtr specifies the segment:offset of the VBE memory windowing function. The windowing function can be invoked either through VBE Function 05h, or by calling the function directly. A direct call will provide faster access to the hardware paging registers than using VBE Function 05h, and is intended to be used by high performance applications. If this field is NULL,

then VBE Function 05h must be used to set the memory window when paging is supported. This direct call method uses the same parameters as VBE Function 05h including AX and for VBE 3.0 implementations will return the correct Return Status. VBE 1.2 implementations and earlier, did not require the Return Status information to be returned. For more information on the direct call method, see the notes in VBE Function 05h and the sample code in Appendix 5. When function 4F01h is called via the protected mode entry point, this pointer is undefined and is not supported.

The **XResolution** and **YResolution** specify the width and height in pixel elements or characters for this display mode. In graphics modes, these fields indicate the number of horizontal and vertical pixels that may be displayed. In text modes, these fields indicate the number of horizontal and vertical character positions. The number of pixel positions for text modes may be calculated by multiplying the returned XResolution and YResolution values by the character cell width and height indicated in the XCharSize and YCharSize fields described below.

The **XCharSize** and **YCharSize** specify the size of the character cell in pixels. This value is not zero based (e.g. XCharSize for Mode 3 using the 9 point font will have a value of 9).

The **NumberOfPlanes** field specifies the number of memory planes available to software in that mode. For standard 16-color VGA graphics, this would be set to 4. For standard packed pixel modes, the field would be set to 1. For 256-color non-chain-4 modes, where you need to do banking to address all pixels, this value should be set to the number of banks required to get to all the pixels (typically this would be 4 or 8).

The **BitsPerPixel** field specifies the total number of bits allocated to one pixel. For example, a standard VGA 4 Plane 16-color graphics mode would have a 4 in this field and a packed pixel 256-color graphics mode would specify 8 in this field. The number of bits per pixel per plane can normally be derived by dividing the BitsPerPixel field by the NumberOfPlanes field.

The **MemoryModel** field specifies the general type of memory organization used in this mode. The following models have been defined:

00h	=	Text mode
01h	=	CGA graphics
02h	=	Hercules graphics
03h	=	Planar
04h	=	Packed pixel
05h	=	Non-chain 4, 256 color
06h	=	Direct Color
07h	=	YUV
08h-0Fh	=	Reserved, to be defined by VESA
10h-FFh	=	To be defined by OEM

Note: VBE Version 1.1 and earlier defined Direct Color graphics modes with pixel formats 1:5:5:5, 8:8:8, and 8:8:8:8 as a Packed Pixel model with 16, 24, and 32-bits per pixel, respectively. In VBE Version 1.2 and later, the Direct Color modes use the Direct Color memory model and use the MaskSize and FieldPosition fields of the ModeInfoBlock to

describe the pixel format. BitsPerPixel is always defined to be the total memory size of the pixel, in bits.

NumberOfBanks. This is the number of banks in which the scan lines are grouped. The quotient from dividing the scan line number by the number of banks is the bank that contains the scan line and the remainder is the scan line number within the bank. For example, CGA graphics modes have two banks and Hercules graphics mode has four banks. For modes that don't have scanline banks (such as VGA modes 0Dh-13h), this field should be set to 1.

The **BankSize** field specifies the size of a bank (group of scan lines) in units of 1 KB. For CGA and Hercules graphics modes this is 8, as each bank is 8192 bytes in length. For modes that do not have scanline banks (such as VGA modes 0Dh-13h), this field should be set to 0.

The **NumberOfImagePages** field specifies the "total number minus one (-1)"of complete display images that will fit into the frame buffer memory. The application may load more than one image into the frame buffer memory if this field is non-zero, and move the display window within each of those pages. This should only be used for determining the additional display pages which are available to the application. Note that for VBE 3.0 implementations, this value will be the *maximum* number of image pages available for both banked and linear modes. To find the actual number of image pages available for either banked or linear modes, please refer to the BnkNumberOfImagePages and LinNumberOfImagepages.

Note: Many applications assume the size of an image page is always rounded up to a 64Kb boundary, so the value returned in NumberOfImagePages should be computed by rounding the size of an image page up to the next 64Kb increment. If this is not done, some known applications may end up writing past the end of available video memory, causing the system to become unstable.

Note: If the ModeInfoBlock is for an IBM Standard VGA mode and the NumberOfImagePages field contains more pages than would be found in a 256KB implementation, the TTY support described in the ModeAttributes must be accurate. i.e., if the TTY functions are claimed to be supported, they must be supported in all pages, not just the pages normally found in the 256KB implementation.

The **Reserved** field has been defined to support a future VBE feature and will always be set to one in this version.

The **RedMaskSize, LinRedMaskSize, GreenMaskSize, LinGreenMaskSize, BlueMaskSize, LinBlueMaskSize, RsvdMaskSize and LinRsvdMaskSize** fields define the size, in bits, of the red, green, and blue components of a direct color pixel. A bit mask can be constructed from the MaskSize fields using simple shift arithmetic. For example, the MaskSize values for a Direct Color 5:6:5 mode would be 5, 6, 5, and 0, for the red, green, blue, and reserved fields, respectively. Note that in the YUV MemoryModel, the red field is used for V, the green field is used for Y, and the blue field is used for U. The MaskSize fields should be set to 0 in modes using a memory model that does not have pixels with component fields. For VBE 3.0 implementations it

is possible to have different color formats in banked framebuffer and linear framebuffer modes. The standard values list the framebuffer format for banked framebuffer modes. For linear modes please refer to the LinXX variants.

The **RedFieldPosition, LinRedFieldPosition, GreenFieldPosition, LinGreenFieldPosition, BlueFieldPosition, LinBlueFieldPosition, RsvdFieldPosition and LinRsvdFieldPosition** fields define the bit position within the direct color pixel or YUV pixel of the least significant bit of the respective color component. A color value can be aligned with its pixel field by shifting the value left by the FieldPosition. For example, the FieldPosition values for a Direct Color 5:6:5 mode would be 11, 5, 0, and 0, for the red, green, blue, and reserved fields, respectively. Note that in the YUV MemoryModel, the red field is used for V, the green field is used for Y, and the blue field is used for U. The FieldPosition fields should be set to 0 in modes using a memory model that does not have pixels with component fields. For VBE 3.0 implementations it is possible to have different color formats in banked framebuffer and linear framebuffer modes. The standard values list the framebuffer format for banked framebuffer modes. For linear modes please refer to the LinXX variants.

The **DirectColorModeInfo** field describes important characteristics of direct color modes. Bit D0 specifies whether the color ramp of the DAC is fixed or programmable. If the color ramp is fixed, then it can not be changed. If the color ramp is programmable, it is assumed that the red, green, and blue lookup tables can be loaded by using VBE Function 09h (it is assumed all color ramp data is 8 bits per primary). Bit D1 specifies whether the bits in the Rsvd field of the direct color pixel can be used by the application or are reserved, and thus unusable.

D0	=	Color ramp is fixed/programmable
	0 =	Color ramp is fixed
	1 =	Color ramp is programmable
D1	=	Bits in Rsvd field are usable/reserved
	0 =	Bits in Rsvd field are reserved
	1 =	Bits in Rsvd field are usable by the application

The **PhysBasePtr** is a 32-bit physical address of the start of frame buffer memory when the controller is in flat frame buffer memory mode. If this mode is not available, then this field will be zero. Note that the physical address cannot be used directly by the application, but must be translated by an operating system service to a linear address that can be used directly by the application (ie: the OS must create the page tables to map in this memory). Under a DPMI compatible environment this is done with DPMI function 0x800. Note also that it is possible for the linear framebuffer memory to start at different locations for different modes.

The **LinBytesPerScanLine** field specifies how many full bytes are in each logical scanline for linear framebuffer modes if the linear framebuffer modes are different to the banked modes. The logical scanline could be equal to or larger than the displayed scanline. VBE 3.0 applications should look at this value for linear modes, as it is possible for the linear modes to have a different logical scanline width than the banked modes.

The **BnkNumberOfImagePages** field specifies the "total number minus one (-1)"of complete display images that will fit into the frame buffer memory for the banked framebuffer version of the video mode. The application may load more than one image into the frame buffer memory if this field is non-zero, and move the display window within each of those pages.

The **LinNumberOfImagePages** field specifies the "total number minus one (-1)"of complete display images that will fit into the frame buffer memory for the linear framebuffer version of the video mode. This is the linear framebuffer version of the above BnkNumberOfImagePages value.

The **MaxPixelClock** field is a 32-bit value that specifies the maximum possible pixel clock that can be selected in this graphics mode when a refresh controlled mode is selected, in units of Hz. Any attempt to select a higher pixel clock will cause the mode set to fail. This field can be used to determine what the maximum available refresh rate for the graphics mode will be.

Note: Version 1.1 and later VBE will zero out all unused fields in the Mode Information Block, always returning exactly 256 bytes. This facilitates upward compatibility with future versions of the standard, as any newly added fields will be designed such that values of zero will indicate nominal defaults or non-implementation of optional features. (For example, a field containing a bit-mask of extended capabilities would reflect the absence of all such capabilities.) Applications that wish to be backwards compatible to Version 1.0 VBE should pre-initialize the 256 byte buffer before calling the Return VBE Mode Information function.

Function 02h - Set VBE Mode

Input:	AX	= 4F02h	Set VBE Mode
	BX	=	Desired Mode to set
		D0-D8=	Mode number
		D9-D10 =	Reserved (must be 0)
		D11 = 0	Use current default refresh rate
		= 1	Use user specified CRTC values for refresh rate
		D12-13	Reserved for VBE/AF (must be 0)
		D14 = 0	Use windowed frame buffer model
		= 1	Use linear/flat frame buffer model
		D15 = 0	Clear display memory
		= 1	Don't clear display memory
	ES:DI	=	Pointer to CRTCInfoBlock structure
Output:	AX	=	VBE Return Status

Note: All other registers are preserved.

This required function initializes the controller and sets a VBE mode. The format of VESA VBE mode numbers is described earlier in this document. If the mode cannot be set, the BIOS should leave the graphics environment unchanged and return a failure error code.

If the requested mode number is not available, then the call will fail, returning AH=01h to indicate the failure to the application.

If bit D11 is set, the mode will be using the CRTC parameters and pixel clock values passed in the CRTCInfoBlock structure, rather than using the default values that the BIOS is currently configured for. This allows the application program or operating system drivers to calculate a new set of CRTC values (preferably using the VESA Generalized Timing Formula (GTF) specification) for the mode, and allow the refresh rate to be set to any supported value for the hardware. If bit D11 is not set, the values passed in ES:DI will be ignored.

If bit D14 is set, the mode will be initialized for use with a flat frame buffer model. The base address of the frame buffer can be determined from the extended mode information returned by VBE Function 01h. If D14 is set, and a linear frame buffer model is not available then the call will fail.

If bit D15 is not set, all reported image pages, based on Function 00h returned information NumberOfImagePages, will be cleared to 00h in graphics mode, and 20 07 in text mode. Memory over and above the reported image pages will not be changed. If bit D15 is set, then the contents of the frame buffer after the mode change is undefined. Note, the 1-byte mode numbers used in Function 00h of an IBM VGA compatible BIOS use D7 to signify the same thing as D15 does in this function. If bit D7 is set and the application assumes it is similar to the IBM compatible mode set using VBE Function 02h, the implementation will fail. VBE aware applications must use the memory clear bit in D15.

Note: VBE BIOS 3.0 implementations may not clear the screen if bit D15 is set. VBE implementations should clear the screen if possible to maintain backwards compatibility, but this is not mandatory (for some controllers this may be difficult from real mode code if the graphics mode is linear only, as the hardware linear framebuffer will not be accessible from real mode code). Hence VBE 3.0 aware applications should always assume that the screen may not have been cleared, and clear it before use.

Note: The VBE 3.0 protected mode entry point may not provide support for extended text modes nor Standard VGA graphics modes. Any attempt to set one of these modes via the protected mode entry point will produce undefined results.

The CRTC information block has the following structure:

```
CRTCInfoBlock struc

HorizontalTotal     dw    ?       ; Horizontal total in pixels
HorizontalSyncStart dw    ?       ; Horizontal sync start in pixels
HorizontalSyncEnd   dw    ?       ; Horizontal sync end in pixels
VerticalTotal       dw    ?       ; Vertical total in lines
VerticalSyncStart   dw    ?       ; Vertical sync start in lines
VerticalSyncEnd     dw    ?       ; Vertical sync end in lines
Flags               db    ?       ; Flags (Interlaced, Double Scan etc)
PixelClock          dd    ?       ; Pixel clock in units of Hz
RefreshRate         dw    ?       ; Refresh rate in units of 0.01 Hz

Reserved            db    40 dup (?)  ; remainder of ModeInfoBlock
CRTCInfoBlock ends
```

The **HorizontalTotal, HorizontalSyncStart, HorizontalSyncEnd, VerticalTotal, VerticalSyncStart** and **VerticalSyncEnd** fields define the default normalized CRTC values that will be programmed if bit D13 is set to a 1. Note that the CRTC values are *normalized* values that are always represented in pixels and lines, rather than in the format used to actually program the hardware. Hence for a 640x480 graphics mode the CRTC values will always be the same regardless of color depth. It is up to the VBE implementation to convert the values from the normalized form to hardware specific values depending on the color depth and other hardware specific requirements. Note also that the CRTC table does not contain any information about the horizontal and vertical blank timing positions. It is up to the VBE implementation to determine the correct blank timings to use for the mode when it is initialized depending on the constraints of the underlying hardware.

The **Flags** field defines the following flags that modify the operation of the mode as follows:

D0	=	Double Scan Mode Enable
	0 =	Graphics mode is not double scanned
	1 =	Graphics mode is double scanned
D1	=	Interlaced Mode Enable
	0 =	Graphics mode is non-interlaced
	1 =	Graphics mode is interlaced
D2	=	Horizontal sync polarity
	0 =	Horizontal sync polarity is positive (+)
	1 =	Horizontal sync polarity is negative (-)
D3	=	Vertical sync polarity
	0 =	Vertical sync polarity is positive (+)
	1 =	Vertical sync polarity is negative (-)

Bit D0 is used to determine whether the mode programmed into the hardware is double scanned or not. When double scanning is specified, the vertical CRTC values passed in will be double what the real vertical resolution will be. Double scanning is used to implement the 200, 240 and 300 line graphics modes on modern controllers. Note that you must check the Double Scan bit in the mode info block to determine if double scan mode is supported before attempting to initialize a double scanned mode. Note also that all 200, 240 and 300 scanline modes require the double scan bit to be set.

Bit D1 is used to determine whether the mode programmed into the hardware is interlaced or non-interlaced. The CRTC timings passed in will be identical for both interlaced and non-interlaced modes, and it is up to the VBE implementation to perform any necessary scaling between the hardware values and the normalized CRTC values in interlaced modes. Note that you must check the Interlaced bit in the mode info block to determine if interlaced mode is supported before attempting to initialize an interlaced mode.

Bit D2 is used to determine the horizontal sync polarity for the CRTC values programmed into the hardware. A value of 0 indicated a positive sync polarity, while a value of 1 indicates a negative sync polarity.

Bit D3 is used to determine the vertical sync polarity for the CRTC values programmed into the hardware. A value of 0 indicated a positive sync polarity, while a value of 1 indicates a negative sync polarity.

The **PixelClock** field defines the *normalized* pixel clock that will be programmed into the hardware. This value is represented in a 32 bit dword in units of Hz. For example to represent a pixel clock of 25.18Mhz one would code a value of 25,180,000. Note that this is the *normalized* pixel clock that will be programmed, not a physical pixel clock and does not include any scaling factors for the mode in question. If the hardware needs the pixel clock to be scaled from the normalized value this will be done by the VBE implementation internally. The normalized pixel clock is necessary in order to be able to calculate the refresh rate for the specific graphics mode using the following formula:

$$refreshRate = \frac{PixelClock}{HorizontalTotal \times VerticalTotal}$$

For example a 1024x768 mode with a HTotal of 1360, VTotal of 802, a normalized pixel clock of 75Mhz might be computed as follows:

$$refreshRate = \frac{65{,}000{,}000}{1360 \times 802} = 59.59\,Hz$$

The **RefreshRate** field defines the refresh rate that the CRTC table defines. This value may not actually be used by the BIOS but *must* be calculated by the application program using the above formulas before initializing the mode. This entry may be used by the BIOS to identify any special cases that may need to be handled when setting the mode for specific refresh rates. The value in this field should be represented in units if .01 Hz (ie: a value 7200 represents a refresh rate of 72.00Hz).

Note:	VBE BIOS 2.0 or later implementations should also update the BIOS Data Area 40:87 memory clear bit so that VBE Function 03h can return this flag. VBE BIOS 1.2 and earlier implementations ignore the memory clear bit.

Note:	This call should not set modes not listed in the list of supported modes. The exception to this is Standard VGA modes and 7 bit mode numbers, which may be initialized regardless of whether the mode is listed and there is a ModeInfoBlock associated with the mode. This is in contrast to VBE/Core 2.0 which requires 7 bit modes to have a ModeInfoBlock associated with the mode.

Function 03h - Return Current VBE Mode

Input:	AX	= 4F03h	Return current VBE Mode
Output:	AX	=	VBE Return Status
	BX	=	Current VBE mode
		D0-D13 =	Mode number
		D14 = 0	Windowed frame buffer model
		= 1	Linear/flat frame buffer model
		D15 = 0	Memory cleared at last mode set
		= 1	Memory not cleared at last mode set

Note: All other registers are preserved.

This required function returns the current VBE mode. The format of VBE mode numbers is described earlier in this document.

Version 1.x Note: In a standard VGA BIOS, Function 0Fh (Read current video state) returns the current graphics mode in the AL register. In D7 of AL, it also returns the status of the memory clear bit (D7 of 40:87). This bit is set if the mode was set without clearing memory. In this VBE function, the memory clear bit will not be returned in BX since the purpose of the function is to return the video mode only. If an application wants to obtain the memory clear bit, it should call the standard VGA BIOS Function 0Fh.

Version 2.x Note: Unlike version 1.x VBE implementations, the memory clear flag will be returned. The application should NOT call the standard VGA BIOS Function 0Fh if the mode was set with VBE Function 02h.

Note: The mode number returned must be the same mode number used in the VBE Function 02h mode set.

Note: This function is not guaranteed to return an accurate mode value if the mode set was not done with VBE Function 02h.

Note: This function is not supported via the VBE 3.0 protected mode entry point, and the results are undefined if it is called.

Function 04h - Save/Restore State

Input:	AX	= 4F04h	Save/Restore State
	DL	= 00h	Return Save/Restore State buffer size
		= 01h	Save state
		= 02h	Restore state
	CX	=	Requested states
		D0=	Save/Restore controller hardware state
		D1=	Save/Restore BIOS data state
		D2=	Save/Restore DAC state
		D3=	Save/Restore Register state
	ES:BX	=	Pointer to buffer (if DL <> 00h)
Output:	AX	=	VBE Return Status
	BX	=	Number of 64-byte blocks to hold the state buffer (if DL=00h)

Note: All other registers are preserved.

This required function provides a complete mechanism to save and restore the display controller hardware state. The functions are a superset of the three subfunctions under the standard VGA BIOS Function 1Ch (Save/restore state) which does not guarantee that the extended registers of the video device are saved or restored. The complete hardware state (except frame buffer memory) should be saveable/restorable by setting the requested states mask (in the CX register) to 000Fh.

Function 05h - Display Window Control

Input:	AX	= 4F05h	VBE Display Window Control
(16-bit)	BH	= 00h	Set memory window
		= 01h	Get memory window
	BL	=	Window number
		= 00h	Window A
		= 01h	Window B
	DX	=	Window number in video memory in window granularity units (Set Memory Window only)
Output:	AX	=	VBE Return Status
	DX	=	Window number in window granularity units (Get Memory Window only)
Input:	BH	= 00h	Set memory window
(32-bit)	BL	=	Window number
		= 00h	Window A
		= 01h	Window B
	DX	=	Window number in video memory in window granularity units (Set Memory Window only)
	ES	=	Selector for memory mapped registers

This required function sets or gets the position of the specified display window or page in the frame buffer memory by adjusting the necessary hardware paging registers. To use this function properly, the software should first use VBE Function 01h (Return VBE Mode information) to determine the size, location and granularity of the windows.

For performance reasons, it may be more efficient to call this function directly, without incurring the INT 10h overhead. VBE Function 01h returns the real mode segment:offset of this windowing function that may be called directly for this reason (for 32-bit protected mode refer to VBE Function 0Ah for obtaining a 32-bit protected mode version of this function). Note that a different entry point may be returned based upon the selected mode. Therefore, it is necessary to retrieve this segment:offset specifically for each desired mode. Note also that the direct call version of this function is not available via the 16-bit protected mode entry point.

Application Developer's Note: This function is not intended for use in a linear frame buffer mode, if this function is requested, the function call will fail with the VBE Completion code AH=03h.

VBE BIOS Implementation Note: If this function is called while in a linear frame buffer memory model, this function must fail with completion code AH=03h.

Note: In VBE 1.2 implementations, the direct far call version returns no Return Status information to the application. Also, in the far call version, the AX and DX registers will be destroyed. Therefore if AX and/or DX must be preserved, the application must do so prior to making the far call. The application must still load the input arguments in BH, BL, and DX (for Set Window). In VBE 3.0 implementations, the BIOS will return the correct Return Status, and therefore the application must assume that AX and DX will be destroyed.

Note: In VBE 2.0 and later implementations the 32-bit version returns no Return Status information to the application. Also note that the ES selector only needs to be passed if the memory location in the Ports and Memory table returned by function 4F0Ah is not defined.

Note: Subfunction 01h (Get Memory Window) is not supported via the VBE 3.0 protected mode entry point, and the results are undefined if it is called.

Function 06h - Set/Get Logical Scan Line Length

Input:	AX	= 4F06h	VBE Set/Get Logical Scan Line Length
	BL	= 00h	Set Scan Line Length in Pixels
		= 01h	Get Scan Line Length
		= 02h	Set Scan Line Length in Bytes
		= 03h	Get Maximum Scan Line Length
	CX	=	If BL=00h Desired Width in Pixels If BL=02h Desired Width in Bytes (Ignored for Get Functions)
Output:	AX	=	VBE Return Status
	BX	=	Bytes Per Scan Line
	CX	=	Actual Pixels Per Scan Line (truncated to nearest complete pixel)
	DX	=	Maximum Number of Scan Lines

This required function sets or gets the length of a logical scan line. This allows an application to set up a logical display memory buffer that is wider than the displayed area. VBE Function 07h (Set/Get Display Start) then allows the application to set the starting position that is to be displayed.

The desired width in pixels or bytes may not be achievable because of hardware considerations. The next larger value will be selected that will accommodate the desired number of pixels or bytes, and the actual number of pixels will be returned in CX. BX returns a value that when added to a pointer into display memory will point to the next scan line. For example, in VGA mode 13h this would be 320, but in mode 12h this would be 80. DX returns the number of logical scan lines based upon the new scan line length and the total memory installed and usable in this display mode.

This function is also valid in VBE supported text modes. In VBE supported text modes the application should convert the character line length to pixel line length by getting the current character cell width through the XCharSize field returned in ModeInfoBlock, multiplying that times the desired number of characters per line, and passing that value in the CX register. In addition, this function will only work if the line length is specified in character granularity. i.e. in 8 dot modes only multiples of 8 will work. Any value which is not in character granularity will result in a function call failure.

Note: On a failure to set scan line length by setting a CX value too large, the function will fail with error code 02h.

Note: The value returned when BL=03h is the lesser of either the maximum line length that the hardware can support, or the longest scan line length that would support the number of lines in the current video mode. This function should return both the maximum scanline width in bytes (BX) and in pixels (CX).

Note: Subfunction 01h (Get Scanline Length) is not supported via the VBE 3.0 protected mode entry point, and the results are undefined if it is called.

Function 07h - Set/Get Display Start

Input:	AX	= 4F07h	VBE Set/Get Display Start Control
(16-bit)	BH	= 00h	Reserved and must be 00h
	BL	= 00h	Set Display Start
		= 01h	Get Display Start
		= 02h	Schedule Display Start (Alternate)
		= 03h	Schedule Stereoscopic Display Start
		= 04h	Get Scheduled Display Start Status
		= 05h	Enable Stereoscopic Mode
		= 06h	Disable Stereoscopic Mode
		= 80h	Set Display Start during Vertical Retrace
		= 82h	Set Display Start during Vertical Retrace (Alternate)
		= 83h	Set Stereoscopic Display Start during Vertical Retrace
	ECX	=	If BL=02h/82h Display Start Address in bytes
			If BL=03h/83h Left Image Start Address in bytes
	EDX	=	If BL=03h/83h Right Image Start Address in bytes
	CX	=	If BL=00h/80h First Displayed Pixel In Scan Line
	DX	=	If BL=00h/80h First Displayed Scan Line
Output:	AX	=	VBE Return Status
	BH	=	If BL=01h Reserved and will be 0
	CX	=	If BL=01h First Displayed Pixel In Scan Line
			If BL=04h 0 if flip has not occurred, not 0 if it has
	DX	=	If BL=01h First Displayed Scan Line
Input:	BH	= 00h	Reserved and must be 00h
(32-bit)	BL	= 00h	Set Display Start
		= 80h	Set Display Start during Vertical Retrace
	CX	=	Bits 0-15 of display start address
	DX	=	Bits 16-31 of display start address
	ES	=	Selector for memory mapped registers

This required function selects the pixel to be displayed in the upper left corner of the display. This function can be used to pan and scroll around logical screens that are larger than the displayed screen. This function can also be used to rapidly switch between two different displayed screens for double buffered animation effects.

For the VBE 2.0 32-bit protected mode version, the value passed in DX:CX is the 32 bit offset in display memory, aligned to a plane boundary. For planar modes this means the value is the byte offset in memory, but in 8+ bits per pixel modes this is the offset from the start of memory divided by 4. Hence the value passed in is identical to the value that would be programmed into the standard VGA CRTC start address register. Note that it is up to the protected mode application to keep track of the color depth and scan line length to calculate the new start address. If a value that is out of range is programmed, unpredictable results will occur. For VBE 3.0 the application

program may optionally pass the missing two bits of information in the top two bits of DX, to allow for pixel perfect horizontal panning. For example (32-bit protected mode interface only):

in planar modes:

CX[15:0] = SA[15:0]
DX[15:0] = SA[31:16]

in 8bpp and greater modes:

CX[15:0] = SA[17:2]
DX[13:0] = SA[31:18]
DX[15:14] = SA[1:0]

VBE 3.0 defines seven new subfunctions (02h, 03h, 04h, 05h, 06h, 82h, 83h) to support hardware triple buffering and stereoscopic LC shutter glasses. Functions 02h and 03h schedule a display start address change to occur during the next vertical retrace, and returns immediately. Function 04h can then be used by the application to determine if the scheduled flip has occurred or not, which can be used for hardware triple buffering to avoid writing to the page being displayed by the CRT controller. Functions 04h and 05h are used to enable and disable free running hardware stereoscopic mode. Functions 82h and 83h schedule the display start address change to occur, and then wait until the address has changed before returning.

Note that functions 02h, 03h, 82h and 83h take the display start address as a byte offset in video memory, whereas the VBE 1.2 compatible functions 00h and 80h take an (x,y) pixel coordinate as the starting address in the frame buffer. Functions 02h and 82h are preferable because they allow for correct page flipping operation in all color depths. Functions 00h and 80h have problems in 24bpp modes where each pixel is represented as three bytes, since there are some combinations of (x,y) starting addresses that may not map

For stereoscopic LC shutter glasses support, the hardware must provide support for free running stereoscopic display with the left and right images located consecutively in video memory, or provide support for dual display start addresses that will be swapped on every vertical retrace interval. On all VBE 3.0 implementation that support hardware stereoscopic display, applications can enable free running stereoscopic display by calling function 05h and the hardware will remain in free running stereoscopic display until the application calls function 06h. When function 05h is called, free running stereoscopic display is enabled with the left buffer located first in video memory and the right buffer located immediately following the left buffer in video memory and the hardware will swap between the two images at each vertical retrace in a sequential fashion (ie: left, right, left, right etc). By default when a mode is initialized, free running stereoscopic display is turned off.

For VBE implementations that support dual display start address hardware (VBE mode information ModeAttributes bit 12 is set), the VBE implementation should enable the stereoscopic display start addressing when the application calls either function 03h or function 83h and it will remain in free running stereoscopic display until the application calls function 06h

to disable stereoscopic display. The main difference with this method of stereoscopic display is that the application has complete control over where the buffers are located in video memory, and two display starting addresses are passed into functions 03h/83h for the complete change.

Note also that the Display Start Address passed for subfunctions 02h, 03h, 82h and 83h are defined as a byte address in video memory rather than as a pixel coordinate. This is in contrast to the values passed to the 32-bit version which is a byte address divided by 4, and allows for controllers that can perform hardware scrolling to a byte boundary for pixel perfect horizontal scrolling.

VBE Implementation Note: If the hardware does not support free running stereoscopic page flipping, the VBE implementation should fail functions 03h, 05h, 06h and 83h. If the hardware does not support dual display start addressing, but does support hardware stereoscopic display with consecutive images in video memory, the implementation should fail functions 03h and 83h but support 05h and 06h. If the hardware does not support the returning of status information for the scheduled display start address swap, the VBE implementation should fail function 04h, however it should still implement functions 02h, 03h, 82h and 83h to provide support for pixel perfect smooth scrolling.

Note: In VBE 2.0 and later implementations the 32-bit version returns no Return Status information to the application. Also note that the ES selector only needs to be passed if the memory location in the Ports and Memory table returned by function 4F0Ah is not defined.

Note: This function is also valid in text modes. To use this function in text mode, the application should convert the character coordinates to pixel coordinates by using XCharSize and YCharSize returned in the ModeInfoBlock. If the requested Display Start coordinates do not allow for a full page of video memory or the hardware does not support memory wrapping, the Function call should fail and no changes should be made. As a general case, if a requested Display Start is not available, fail the Function call and make no changes.

Note: CX and DX, for both input and output values are zero-based.

Note: Subfunction 01h (Get Display Start) is not supported via the VBE 3.0 protected mode entry point, and the results are undefined if it is called.

Function 08h - Set/Get DAC Palette Format

Input:	AX	= 4F08h	VBE Set/Get Palette Format
	BL	= 00h	Set DAC Palette Format
		= 01h	Get DAC Palette Format
	BH	=	Desired bits of color per primary (Set DAC Palette Format only)
Output:	AX	=	VBE Return Status
	BH	=	Current number of bits of color per primary

This required function manipulates the operating mode or format of the DAC palette. Some DACs are configurable to provide 6 bits, 8 bits, or more of color definition per red, green, and blue primary colors. The DAC palette width is assumed to be reset to the standard VGA value of 6 bits per primary color during any mode set.

An application can determine if DAC switching is available by querying Bit D0 of the Capabilities field of the VbeInfoBlock structure returned by VBE Function 00h (Return Controller Information). The application can then attempt to set the DAC palette width to the desired value. If the display controller hardware is not capable of selecting the requested palette width, then the next lower value that the hardware is capable of will be selected. The resulting palette width is returned.

This function will return failure code AH=03h if called in a direct color or YUV mode.

Note: Subfunction 01h (Get DAC Palette Format) is not supported via the VBE 3.0 protected mode entry point, and the results are undefined if it is called.

Function 09h - Set/Get Palette Data

Input:	AX	= 4F09h	VBE Load/Unload Palette Data
(16-bit)	BL	= 00h	Set Palette Data
		= 01h	Get Palette Data
		= 02h	Set Secondary Palette Data
		= 03h	Get Secondary Palette Data
		= 80h	Set Palette Data during Vertical Retrace
	CX	=	Number of palette registers to update (to a maximum of 256)
	DX	=	First of the palette registers to update (start)
	ES:DI	=	Table of palette values (see below for format)
Output:	AX	=	VBE Return Status
Input:	BL	= 00h	Set Palette Data
(32-bit)		= 80h	Set Palette Data during Vertical Retrace
	CX	=	Number of palette registers to update (to a maximum of 256)
	DX	=	First of the palette registers to update (start)
	ES:EDI	=	Table of palette values (see below for format)
	DS	=	Selector for memory mapped registers

This required function is very important for NonVGA controllers which don't provide access to the palette via the standard VGA RAMDAC registers. Applications should check the NonVGA bit in the ModeInfoBlock for the mode they are using, and if the mode is VGA compatible then you may program the palette directly via the standard VGA palette registers. If the mode is NonVGA however, then this function must be used.

Note that for VGA compatible controllers, if the palette width has been set to 8 bits per primary rather than the default 6, applications simply need to program a full 8 bits at a time via the standard VGA RAMDAC registers (in Standard VGA 6 bit modes only the bottom 6 bits are used). When in 6 bit mode, the format of the 6 bits is LSB for speed reasons, as the application can typically shift the data faster than the BIOS can.

Setting the palette data during the vertical retrace is required on some older RAMDAC's to avoid the onset of 'snow like' effects on the screen during palette programming. Check bit D2 of the capabilities field returned by VBE Function 00h to determine if this is required by the RAMDAC. Many newer style RAMDAC's don't have this limitation and can easily be programmed at any time. Note however setting the palette data during the vertical retrace is also very useful for doing smooth palette fades and changes so that the palette data is instantly changed before the next frame is displayed (such as done by games when moving from level to level etc).

The palette values passed will be an array of entries with the following structure for each entry in the array:

```
PaletteEntry struc
Blue        db    ?     ; Blue channel value (6 or 8 bits)
Green       db    ?     ; Green channel value (6 or 8 bits)
Red         db    ?     ; Red channel value(6 or 8 bits)
Alignment   db    ?     ; DWORD alignment byte (unused)
PaletteEntry ends
```

Note: In VBE 2.0 and later implementations the 32-bit version returns no Return Status information to the application. Also note that the DS selector only needs to be passed if the memory location in the Ports and Memory table returned by function 4F0Ah is not defined.

Note: The need for the secondary palette is for anticipated future palette extensions, if a secondary palette does not exist in an implementation and these calls are made, the VBE implementation will return error code 02h.

Note: All applications should assume the DAC is defaulted to 6 bit mode. The application is responsible for switching the DAC to higher color modes using Function 08h.

Note: Query VBE Function 08h to determine the RAMDAC width before loading a new palette.

Note: Subfunctions 01h and 03h (Get Palette Data and Get Secondary Palette Data) are no longer mandatory for VBE 3.0, and may return a failure code if called. These functions are also not supported by the protected mode entry point and the results are undefined if they are called.

Function 0Ah - Return VBE Protected Mode Interface

Input:	AX	= 4F0Ah	VBE 2.0 Protected Mode Interface
	BL	= 00h	Return protected mode table
Output:	AX	=	Status
	ES	=	Real Mode Segment of Table
	DI	=	Offset of Table
	CX	=	Length of Table including protected mode code in bytes (for copying purposes)

This optional function call returns a pointer to a table that contains code for a simple 32-bit protected mode interface that can either be copied into local 32-bit memory space or can be executed from ROM providing the calling application sets all required selectors and I/O access correctly. This function returns a pointer (in real mode space) with offsets to the code fragments, and additionally returns an offset to a table which contains Non-VGA Port and Memory locations which an Application may have to have I/O access to. This function currently includes but is not limited to bank switching, display start address programming (for double/triple buffering) and palette programming.

Note: For VBE 3.0 and above this function is optional and may not be implemented. VBE 3.0 applications primarily use the linear framebuffer for speed so 32-bit bank switching functions are not required, and the new protected mode entry point for VBE 3.0 provides high performance access from protected mode applications and operating systems to all the standard VBE functions.

The format of the table is as follows:

ES:DI + 00h	Word Offset in table of Protected mode code for Function 5 for Set Window Call
ES:DI + 02h	Word Offset in table of Protected mode code for Function 7 for set Display Start
ES:DI + 04h	Word Offset in table of Protected mode code for Function 9 for set Primary Palette data
ES:DI + 06h	Word Offset in table of Ports and Memory Locations that the application may need I/O privileges for. (Optional: if unsupported this must be 0000h) (See Sub-table for format)
ES:DI + ?	Variable remainder of Table including Code

The format of the Sub-Table (Ports and Memory locations)

Port, Port, ... , Port, Terminate Port List with FF FF, Memory location base addresses (4 bytes), Limit (ie: length-1) (2 bytes), Terminate Memory Location with FF FF.

Example 1. For Port/Index combination 3DE/Fh and Memory locations DE800-DEA00h (length = 200h) the table would look like this:

```
DE 03 DF 03 FF FF 00 E8 0D 00 00 01 FF FF
```

Example 2. For only the ports it would look like:

```
DE 03 DF 03 FF FF FF FF
```

Example 3. For only the memory locations it would look like

```
FF FF 00 E8 0D 00 00 01 FF FF
```

If the memory location is zero, then only I/O mapped ports will be used so the application does not need to do anything special. This should be the default case for ALL cards that have I/O mapped registers because it provides the best performance and is easier for application programmers to deal with.

If the memory location is nonzero (there can be only one), the application will need to create a new 32-bit selector with the base address that points to the "physical" location specified with the specified limit. This selector must be passed in the ES or DS segment register (depending on the function) so that the 32-bit code can directly access its memory mapped registers as absolute offsets into the ES selector (i.e., mov [es:10],eax to put a value into the register at base+10). For more information on the parameters passed to the 32-bit protected mode functions, please refer to the definitions for each of the functions in question. All the 32-bit protected mode functions that are defined are completely relocateable and can be copied and executed anywhere in 32-bit protected mode code space. However in order to function correctly, there are special calling requirements that must be followed when you call these functions:

- The CS, DS, ES and SS segment registers must be contain valid 32-bit protected mode selectors when the functions are called, and SS:ESP must point to a valid 32-bit stack. Generally the application can simply use the default selectors provided by the 32-bit host environment, except as noted below.

- There are some cases where the protected mode functions may *require* access to memory mapped registers in order to complete the operation. The *physical* location and length of the memory mapped register area that must be mapped will be listed in the Sub-Table (Ports and Memory) returned by VBE Function 0Ah. If the memory location is zero (NULL), then only I/O mapped ports will be used and the application does not need to do anything special (which should be the default case for ALL cards that have I/O mapped registers to provide the best performance). If however the memory location is nonzero, the application *must* create a selector that points to the memory location, and then pass this selector to the calling function in either DS or ES (see the function definitions for the appropriate functions to determine which register it should be passed in). It is up to the application code to save and restore the previous

state of the DS or ES selector (depending in the function being called) if this is necessary

- When the VBE services are called, the current I/O permission bit map must allow access to the I/O ports that the VBE may need to access. This can be found in the Sub-Table (Ports and Memory) returned by VBE Function 0Ah.

- The high order words of any registers that are used to pass parameters to the 32-bit functions may be destroyed by the 32-bit functions.

To summarize, it is the responsibility of the calling application to ensure to that it has the appropriate I/O and memory privileges, and a large enough stack and appropriate selectors allocated. It is also the responsibility of the calling application to preserve registers if necessary.

Note: All protected mode functions should end with a near RET (as opposed to FAR RET) to allow the application software to CALL the code from within the ROM.

Note: The Port and Memory location Sub-table does not include the Frame Buffer Memory location. The Frame Buffer Memory location is contained within the ModeInfoBlock returned by VBE Function 01h.

Note: The protected mode code is assembled for a 32-bit code segment, when copying it, the application must copy the code to a 32-bit code segment.

Note: Currently undefined registers may be destroyed with the exception of ESI, EBP, DS and SS.

Function 0Bh - Get/Set pixel clock

Input:	AX	= 4F0Bh	Get/Set pixel clock
	BL	= 00h	Get closest pixel clock
	ECX	=	Requested pixel clock in units of Hz
	DX	=	Mode number pixel clock will be used with
Output:	AX	=	Status
	ECX	=	Closest pixel clock (BL = 00h)

This required function allows an application to determine if a particular pixel clock is available. When this function is called with BL set to 0, it will run the requested pixel clock through the internal PLL programming routines and return the actual pixel clock that will be programmed into the hardware in the ECX register. The process of running the PLL clock computation routines may cause the returned pixel clock to be rounded slightly up or down from the requested value, however the BIOS should implement the algorithms to attempt to find clocks that are the same as or higher than the requested value. Note that the calling application must also pass in the VBE mode number for the graphics mode that will be using this pixel clock to this function. The mode number is necessary so that the underlying programming code can determine apply any necessary scaling internally before looking for the closest physical pixel clock, and then scaling the result back. This ensures that the application programmer only ever has to deal with normalized pixel clocks and not have to worry about pixel clock scaling issues.

If the BIOS implementation uses a table driven clock programming approach, it should always attempt to find the next highest pixel clock in the table to the requested clock. The exception to this is if there is a lower clock in the table that is within a tolerance of 1% of the requested clock in which case this clock should be returned.

This pixel clock can then be used by the application to compute the exact GTF timing parameters for the mode. Note that for hardware that is not fully programmable, the returned pixel clock that is the closest the one desired may be substantially different (ie: you could get back 25Mhz when you request 29Mhz). It is up the calling application to determine if the clock is suitable and to attempt to choose a different clock if not suitable. The pixel clocks passed in and returned occupy one dword (32 bits) that represents the pixel clock in units of Hz (ie: a pixel clock of 25.18Mhz is represented with a value of 25,180,000).

VBE Supplemental Specifications

This chapter details VBE Supplemental Specifications.

Purpose of Supplemental Specifications

The VBE was originally designed to provide a device-independent interface between application software and SVGA hardware. In the last few years, the personal computing environment has grown much more complex and there have been numerous requests to provide interfaces similar to the VBE to service these new requirements. The VBE supplemental specification architecture provides a way to extend the basic VBE specification without making it too unwieldy or having to revise the VBE specification itself.

The supplemental specifications are implemented using VBE function numbers starting at AL=10h. This leaves the first sixteen functions available for eventual VBE growth. Individual calls for each supplemental specification are made through a subfunction number via the BL register. This function/subfunction architecture is compatible with the VBE and provides each VBE Supplemental Specification with 64 potential subfunctions. Subfunction 00h for each supplemental specification is reserved for a 'Return VBE Supplemental Specification Information' call. It is based on the VBE Function 00h and returns basic information on the VBE Supplemental Specification implementation.

Obtaining Supplemental VBE Function Numbers

VBE Supplemental Specifications can only be created by VESA committees. Once a need for a new software specification has been identified, the group working on it needs to contact the VESA Software Standards Committee (SSC) to discuss the requirements. The SSC will assign a function number and name to the supplemental specification. The name assigned to a supplemental specification will be in the form of 'VBE/???' where the '???' is a two or three letter acronym for its function. Two letter acronyms will be padded with FFh for the third letter. In some cases the committee that is working on the supplemental specification may have another name that they will use for promotional purposes, however the VBE/??? will continue to be the signature.

The VBE specification will be revised periodically to update the list of supplemental specifications. To obtain the actual specifications, contact the VESA office.

Required VBE Supplemental Specification Components

VBE Supplemental Specification Functions

All VBE Supplemental Specification functions are called with the AH register set to 4Fh to identify them as VBE function calls. The AL register is used to specify which VESA function the supplemental specification is using. The BL register will contain the subfunction number for the call being made, BH will contain the sub-function number if necessary.

e.g., Input:	AX	= 4FXXh	VESA Supplemental VBE Specification ('XX' represents the function number for the supplemental specification)
	BL	= ?	Subfunction
	BH	= ?	Sub-function

Return Status

All VBE Supplemental Specifications will use the VBE Completion codes as documented in Section 4.1 of the VBE specification.

Subfunction 00h - Return VBE Supplemental Specification Information

This subfunction returns the capabilities, revision level, and vendor specific information of the supplemental specification, and is a required function for any VBE 2.x Supplemental Specification. Note that supplementation functions provided by VBE 3.0 implementations must be fully protected mode compatible and callable via the protected mode entry point for the VBE 3.0 implementation. This includes the VBE/PM, VBE/DDC and other important supplemental specifications.

The purpose of this subfunction is to provide information to the calling program about the general capabilities of the installed VBE software and hardware. This subfunction fills an information block structure at the address specified by the caller. The SupVBEInfoBlock information block size is 256 bytes.

Input:	AX	= 4FXXh	Return Supplemental VBE Specification Information ('XX' represents the function number for the supplemental specification)
	BL	= 00h	Subfunction '0'
	ES:DI	=	Pointer to buffer in which to place SupVBEInfoBlock structure
Output:	AX	=	VBE Return Status

Other registers may be defined for input and output based upon the particular requirements of supplemental specifications

When writing supplemental functions, explicitly state which registers are preserved and which are destroyed. Refrain from preserving all registers as this tends to limit expandability in the future. An example of the note is:

> **Note:** Currently undefined registers may be destroyed with the exception of SI,BP,DS and SS.

The information block has the following structure:

```
SupVbeInfoBlock struc
SupVbeSignature     db      'VBE/???'       ; Supplemental VBE Signature
SupVbeVersion       dw      ?               ; Supplemental VBE Version
SupVbeSubFunc       db      8 dup (?)       ; Bitfield of supported subfunctions
OemSoftwareRev      dw      ?               ; OEM Software revision
OemVendorNamePtr    dd      ?               ; VbeFarPtr to Vendor Name String
OemProductNamePtr   dd      ?               ; VbeFarPtr to Product Name String
OemProductRevPtr    dd      ?               ; VbeFarPtr to Product Revision String
OemStringPtr        dd      ?               ; VbeFarPtr to OEM String
Reserved            db      221 dup (?)     ; Reserved for description strings and future
                                            ; expansion
SupVbeInfoBlock ends
```

> **Note:** All data in this structure is subject to change by the VBE implementation when any VBE Subfunction 00h is called. Therefore it should not be used by the application to store data of any kind.

Description of the **SupVbeInfoBlock** structure fields:

The **SupVbeSignature** field is filled with the ASCII characters 'VBE/' followed by the two or three letter acronym that represents the supplemental specification. This field is filled by the supplementary VBE implementation. In the event that the acronym is only two letters, the third letter must be filled with FFh.

The **SupVbeVersion** is a BCD value which specifies what level of the VBE supplementary specification is implemented in the software. The higher byte specifies the major version number. The lower byte specifies the minor version number.

> **Note:** The BCD value for 2.0 is 0200h and the BCD value for 1.2 is 0102h. In the past we have had some applications misinterpreting these BCD values. For example, BCD 0102h was interpreted as 1.02, which is incorrect.

The **SupVbeSubFunc** is a bitfield that represents the subfunctions available for the supplementary specification. If the bit representing a particular subfunction is set, then that subfunction is supported. Subfunction '0' is represented by the LSB of the first byte and the other subfunctions follow. Only bits for subfunctions defined in the specification need to be set.

The **OemStringPtr** is a VbeFarPtr to a null-terminated OEM-defined string. This string may used to identify the graphics controller chip or OEM product family for hardware specific display drivers. There are no restrictions on the format of the string. This pointer may point into either ROM or RAM, depending on the specific implementation.

The **OemSoftwareRev** field is a BCD value which specifies the OEM revision level of the Supplemental Specification software. The higher byte specifies the major version number. The lower byte specifies the minor version number. This field can be used to identify the OEM's VBE software release.

The **OemVendorNamePtr** is a VbeFarPtr to a null-terminated OEM-defined string containing the name of the vendor who produced the display controller board product.

The **OemProductNamePtr** is a VbeFarPtr to a null-terminated OEM-defined string containing the product name of the display controller board.

The **OemProductRevPtr** is a VbeFarPtr to a null-terminated OEM-defined string containing the revision or manufacturing level of the display controller board product. This field can be used to determine which production revision of the display controller board is installed.

Loading Supplemental Drivers

VBE Supplemental Specifications can be implemented in ROM, TSR programs or as device drivers. The specific requirements will vary depending on the individual supplementary specification. If there are any specific requirements, they should be detailed in the supplementary specification.

Implementation Questions

When developing a new supplemental specification, implementation questions whether they are covered in this guideline or not, should be referred to the VESA Software Standards Committee for clarification. The chairman of the SSC can be contacted through the VESA office.

Known Supplemental Specifications

Function 10h - Power Management Extensions (PM)

This optional function controls the power state of the attached display device or monitor. Refer to the VBE/PM Standard for specifics.

Function 11h - Flat Panel Interface Extensions (FP)

This proposed optional supplemental specification allows access to the special features incorporated in Flat Panel controllers. There is no reference specification at the time of this standard's approval. Contact the VESA office for more information.

Function 13h - Audio Interface Extensions (AI)

This optional function provides standard Audio services.
Refer to the VBE/AI Standard for specifics.

Function 14h - OEM Extensions

This optional supplemental function provides OEM's with a code dispatch area that falls under the VESA 4Fh functions. An OEM may use this area at their own risk. VESA states no warranties or guarantees about the function calls contained within this area.

Function 15h - Display Data Channel (DDC)

This optional function provides a mechanism to extract data from attached display devices on the VESA communication channel. Refer to the VBE/DDC Standard for specifics.

Appendix 1 - VBE Implementation Considerations

This appendix discusses required features of VBE 3.0. implementations, and offers some suggestions for consideration by BIOS developers. Some issues raised here apply only to adding VBE 3.0 to an existing VGA BIOS, while other issues are more generally relevant.

Minimum Functionality Requirements

Required VBE Services

VBE Functions 00h-0Ah are required; all other functions are optional. There are no absolutely required modes, or mode capabilities, since these will vary according to the hardware and applications.

Note:	With VBE 3.0 function 4F0Ah is optional as the new protected mode entry point provides similar functionality with reduced BIOS space. Also function 4F09h is optional if and only if the controller is fully VGA compatible, in which case the application can program the palette directly using the standard VGA palette registers. For NonVGA compatible controllers this function is required to properly support palette modes. Note also that these functions must be implemented, but implementations may simply fail the functions if not required.

Minimum ROM Implementation

For compliance certification, Functions 00-0Ah must be implemented in the ROM. In the case of ROM space limitations that do not allow full implementations, VESA strongly recommends that VBE Get Controller Information Function 00h be implemented in the ROM so applications will be able to find information about the controller type and capabilities. This 'Stub' implementation can be supplemented by a TSR which will provide full VBE Core functionality. These stub implementations are not VBE 3.0 compliant and should only be implemented in cases where no space is available to implement the whole VBE. In the event that a stub is implemented a TSR must be available to complete VBE 3.0 functionality.

In a stub implementation, the VideoModeList will contain no entries (starts with 0FFFFh). This is the indicator to application software that the VBE Core implementation is in fact only a stub and that other functions and modes do not exist.

TSR Implementations

TSR based implementations of VBE must not assume that a compatible graphics controller is present! They must first attempt to detect the presence of a compatible device before chaining into INT 10h and completing the load process. If no compatible hardware is detected they must

exit without chaining INT 10h. On failure to load, the TSR should display an appropriate message to the screen, identifying both the installed and expected hardware and displaying the OEM strings from Function 00h, if available. The software version number and identifying information for the TSR should also be shown.

TSRs which are meant to work in a variety of hardware and BIOS environments should check to see if the ROM supports some version of VBE Function 00h, Get Controller Information. The information which is returned from this function can then be passed on to calling applications or displayed on the screen, reducing the burden of supporting different display hardware. If a stub or incomplete version of VBE exists in ROM, it is the responsibility of the TSR to supplement all missing functions and replace Function 00h.

VESA recommends that VBE 3.0 TSRs be given names which contain some identifier for the OEM and/or product such as ATIVBE.EXE or S3VBE.EXE. This will help users make sure they have the correct version of software for their hardware, and may prevent a few phone calls for software support. It is also required that a help screen be included, which can be activated by typing "/h", "/?", or any unrecognized parameter on the command line. The help screen should contain all pertinent information about the source and version number of the TSR and the hardware on which it is designed to work.

VGA BIOS Implications

A primary design goal with the VESA VBE is to minimize the effects on the standard VGA BIOS. Standard VGA BIOS functions should need to be modified as little as possible. However, two standard VGA functions are affected by the VBE. These are VGA Function 00h (Set VGA Mode) and VGA Function 0Fh (Get Current VGA Mode).

VBE-unaware applications (such as old Pop-Up programs and other TSRs, or the CLS command of MS-DOS), may use VGA BIOS Function 0Fh to get the current display mode and later call VGA BIOS Function 00h to restore/reinitialize the old graphics mode. To make such applications work, the 8-bit value returned by VGA BIOS Function 0Fh (it is up to the OEM to define this number), must correctly reinitialize the graphics mode through VGA BIOS Function 00h.

However, VBE-aware applications should not set the VBE mode using VGA BIOS Function 00h, or get the current mode with VGA BIOS Function 0Fh. VBE Functions 02h (Set VBE mode) and 03h (Get VBE Mode) should be used instead. The mode number must be from the mode list returned by VBE Function 00h, and Function 03h must return the same mode number used to set the mode in Function 02h.

Given these requirements, and the fact that many BIOS manufacturers will need to support at least some of the VESA-defined 14-bit mode numbers for backwards compatibility, it is clear that the BIOS must keep track of the last mode that was set with VBE Function 02h. There are various ways that this could be accomplished without the use of scratch registers or non-volatile RAM, which is not always available. One method is to use the mode number byte in the BIOS

Data Area to store the index into the mode list returned in VBE Function 00h, which is always stored in ROM. Another method is to store a small translation table for the 14-bit mode numbers (probably necessary for using duplicate mode numbers anyway) and to use an obsolete or unused bit in the BIOS Data Area to indicate a 14-bit mode in effect.

If a BIOS offers only the flat frame buffer version of one of the modes which have VESA-defined numbers, it may be advisable to use an OEM-defined number for that mode instead. Since VBE 1.2 and earlier versions assume standard VGA windowing of the frame buffer, older VBE-aware applications may recognize the mode number and attempt to use windowed memory without properly checking with Function 01h.

Real mode ROM Space Limitations

Since standard VGA BIOS is currently confined to 32K ROM images, space is likely to be critical in implementing even the minimum VBE 3.0 functionality. Most VGA BIOSs have already been compressed many times as new features and modes have been added over time. Clearly, older VGA BIOS features may have to be sacrificed to make room.

Data Storage

To allow for ROM based execution of the VBE functions, each VBE function must be implemented without the use of any local data. When possible, the BIOS data area, non-volatile RAM, or OEM specific scratch registers can be used to place scratch data during execution. All VBE data structures are allocated and provided to VBE by the calling application.

Removal of Unused VGA Fonts

VESA strongly recommends that removal of the 8x14 VGA font become a standard way of freeing up space for VBE 3.0 implementations. The removal of this font leaves 3.5K bytes of ROM space for new functions, and is probably the least painful way to free up such a large amount of space while preserving as much backwards compatibility as possible. The 8x14 font is normally used for VGA Modes 0*, 3* and Mode 10h, which are 350-line or EGA compatible modes. When these modes are selected the 8x16 font may be chopped and used instead. When chopping a 16 point font to replace the 14 point, there are several characters (ones with descenders) that should be special cased.

Some applications which use the 8x14 font obtain a pointer to the font table through the standard VGA functions and then use this table directly. In such cases, no workaround using the 8x16 font is possible and a TSR with the 8x14 font is unavoidable. Some OEMs may find this situation unacceptable because of the potential for an inexperienced user to encounter "garbage" on the screen if the TSR is not present. However, OEMs may also find eventually that demand for VBE 3.0 services is great enough to justify the inconvenience associated with an 8x14 font TSR. To date, no compatibility problems are known to be caused by the use of such a TSR. VESA will make available a TSR that replaces the 8x14 font, please contact VESA for more information.

Another option with the fonts in Turn-Key systems (such as Laptops, Notebooks etc.) is to move the fonts to another location in the System ROM. In fact VBE functions could even be relocated. This however is not an acceptable solution for most desktop systems, where they are expandable.

Deleting VGA Parameter Tables

One way to create more ROM space for the VBE is to delete some of the VGA parameter tables by deleting modes which are outdated and little used. Many of the standard VGA modes are now almost entirely obsolete and should probably be phased out of existence. How quickly this might happen depends on which applications are still using the older modes and on how tolerant OEMs and users will be to using TSR programs for these modes when necessary. Some mode groups which might be candidates for removal are modes 4, 5, and 6, all CGA modes, or all 200 line modes.

It must be emphasized, however, that it is absolutely necessary to preserve the size and positions of all the standard mode VGA parameter tables! Failure to do so will cause a lot of problems with diagnostics and older VGA applications. If a table is removed, fill the space with an equal number of bytes of code or data.

Increasing ROM Space

In the PC environment, VGA BIOS developers have traditionally been limited to a 32K ROM image located at C0000h-C7FFFh. The C8000h-CBFFFh area was originally reserved for the XT hard disk BIOS, which is of little current concern. However, SCSI CD ROM controllers have now begun to use this area, and the possibility exists that other devices may use this area also. It is unlikely that VBE developers will be able to expand into the C8000h-CFFFFh region without creating potential conflicts.

Support of VGA TTY Functions

The support of VGA TTY functions is recommended, but not mandatory, for graphic modes beyond VGA. TTY support for all modes is desirable to allow basic text operations such as reading and writing characters to the screen. Some operating systems will revert to using TTY functions when a hardware error occurs, since the graphics environment may no longer be operational.

Support of TTY functions for all modes will, of course, increase the size of the BIOS. One possible solution is to provide TTY function support for extended modes as part of a TSR rather than in the ROM.

Bit D2 in the Mode Attributes field in the ModeInfoBlock structure returned by VBE Function 01h indicates the presence of support for TTY functions for each VBE mode. Refer to the VBE Function 01h description for details on which TTY functions must be supported when this bit is set.

Developing Dual-Mode BIOS code

In order to support the Protected Mode Entry point for VBE 3.0 compatibility, the 16-bit BIOS code must be crafted as 'dual-mode' code. This essentially means that the code is designed to run as either 16-bit real mode code or 16-bit protected mode code without modification.

When the BIOS code is called from protected mode, the application or OS will have modified the PMInfoBlock to contain valid selectors to all the important physical memory regions that the BIOS may need to access, along with a selector to a writeable BIOS data area block. In your BIOS code, rather than directly loading selector values to access physical memory regions or the BIOS data area, load the selector values directly from the PMInfoBlock in the BIOS. This way the same code will run identically as real mode code or as 16-bit protected mode code.

For instance, instead of loading the selectors as follows:

```
xor  ax,ax
mov  ds,ax          ; DS points to BIOS data area
mov  ax,A000h
mov  ds,ax          ; DS points to VGA framebuffer
```

you can load the segment registers with the following dual-mode compatible code:

```
mov  ds,[cs:PMInfo.BIOSDataSel]
mov  ds,[cs:PMInfo.A0000Sel]
```

Determining when in Protected Mode

In some instances it may be necessary for the BIOS code to determine if it is running in real mode or in protected mode. Before the BIOS code is called from protected mode, the application will set the InProtectMode field of the PMInfoBlock to a '1', indicating that the code is now running in protected mode. By default the value of this field will be set to 0 in the real mode BIOS, so the BIOS code can simply look at this field to determine the mode it is running in.

Things to avoid in Protected Mode

If it is necessary for the BIOS code to directly access PCI configuration space registers, access to these registers is not possible via the INT 1Ah software interrupt that is normally used from real mode code. In order to fully support the protected mode entry point, the BIOS implementer cannot call the INT 1Ah software interrupt, but must access the PCI config space registers using the standard I/O port access methods as outlined in the PCI 2.1 hardware specification.

When writing dual-mode code you should also avoid the following real mode specific tricks:

- Segment arithmetic, since memory is accessed via selectors not segments.
- Hard coded segment register values. Always load segment registers from the values stored in the PMInfoBlock structure.

Returning pointers in info blocks

When called via the protected mode entry point, all pointers returned must be valid 16:16 protected mode pointers, not real mode pointers. Hence the pointers to the OEMString, VideoModeTable etc cannot have hard coded segment values. You can easily modify your BIOS code to support this with the following steps:

1. If the data is located in the BIOS image, set the segment value to the value of the CS register, rather than directly setting it to C000h. Under real mode CS will contain C000h and under protected mode it will contain a read-only selector that points to the start of the BIOS image.

2. If the data is located in the passed in info block, set the segment value to the ES selector value that is passed in to the function.

Supporting Multiple Controllers

It is sometimes necessary for more than one display controller to be present in the system for several reasons. For example, OEMs may choose to implement a dual-controller design with VGA functionality provided by one controller, and SVGA or VBE functionality provided by a second controller. In some cases, it may be desirable to install more than one display adapter in the system for simultaneous support of multiple display monitors (to support dual monitor debugging for example).

Dual-Controller Designs

VBE 3.0 supports the dual-controller design by assuming that since both controllers are typically provided by the same OEM, under control of a single BIOS ROM on the same graphics card, it is possible to hide the fact that two controllers are indeed present from the application. This has the limitation of preventing simultaneous use of the independent controllers, but allows applications released before VBE 3.0 to operate normally. The VBE Function 00h (Return Controller Information) returns the combined information of both controllers, including the combined list of available modes. When the application selects a mode, the appropriate controller is activated. Each of the remaining VBE functions then operates on the active controller.

Provision for Multiple Independent Controllers

It is not possible to support multiple independent controllers via the VBE/Core interface. However multiple independent controllers can be fully supported using the VBE/AF Accelerator Functions specification (contact VESA for more information), and also allow the graphics accelerator functions for all installed controllers to be programmed concurrently.

OEM Extensions to VBE

The VBE specification allows the OEM to extend its functionality for support of nonstandard, or private features known only to the OEM and custom applications that are aware of these OEM extensions.

VBE Function 14h is reserved for use by OEMs wishing to add VBE subfunctions of their own. This function number is provided so that the OEM may add custom services without fear of conflict with other VBE services. These subfunctions must use the AX register in the same manner as all other standard VBE functions and return the standard VBE completion codes.

Normally, these extended functions are used by the OEM to aid in the setup and configuration of the controller hardware. For example, during installation it may be necessary to set the physical frame buffer address, maximum monitor refresh frequency, default graphics mode, default power state, etc. A single setup and installation program can be used by the OEM with the entire product line if the same OEM extensions are implemented on each product.

A utility accessing these proprietary functions must read the VbeInfoBlock returned by VBE Function 00h to determine if the firmware is of the proper type and revision level before making any Function 14h calls. Failure to do so will render the calling utility incompatible with VBE 3.0 and may cause unpredictable results.

Appendix 2 - Sample Source Code

The certification process is only for the BIOS implementations, this should be enough to ensure that the applications fall in line with the VESA standard. If it doesn't work on a VBE Compliant card, then the application is wrong and should be changed. To help ensure that the application developer will work on VBE Compliant systems, sample source for application developer's will be provided by the VESA office.

A simple example of how to set a Video Mode, and how to use it to put something up on the screen, is found below. This is not intended to be a complete SDK or source example, but it only demonstrates what we are trying to achieve.

C Language Module

(This has been compiled and tested under Microsoft C 6.0. Conversion for the direct banking method to inline assembly may be required for Borland C.)

```
/****************************************************************************
*
*                               Hello VBE!
*
*
* Language:     C (Keyword far is by definition not ANSI, therefore
*                  to make it true ANSI remove all far references and
*                  compile under MEDIUM model.)
*
* Environment:  IBM PC (MSDOS) 16 bit Real Mode
* Original code contributed by:     - Kendall Bennett, SciTech Software
* Conversion to Microsoft C by:     - Rex Wolfe, Western Digital Imaging
*                                   - George Bystricky, S-MOS Systems
*
* Description:  Simple 'Hello World' program to initialize a user
*               specified 256 color graphics mode, and display a simple
*               moire pattern. Tested with VBE 1.2 and above.
*
*               This code does not have any hard-coded VBE mode numbers,
*               but will use the VBE 2.0 aware method of searching for
*               available video modes, so will work with any new extended
*               video modes defined by a particular OEM VBE 2.0 version.
*
*               For brevity we don't check for failure conditions returned
*               by the VBE (but we shouldn't get any).
*
****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <conio.h>

/* Comment out the following #define to disable direct bank switching.
 * The code will then use Int 10h software interrupt method for banking. */
```

```
#define  DIRECT_BANKING

#ifdef  DIRECT_BANKING
/* only needed to setup registers BX,DX prior to the direct call.. */
extern far setbxdx(int, int);
#endif

/*---------------------- Macro and type definitions ----------------------*/

/* SuperVGA information block */

struct
{
    char    VESASignature[4];       /* 'VESA' 4 byte signature         */
    short   VESAVersion;            /* VBE version number              */
    char    far *OEMStringPtr;      /* Pointer to OEM string           */
    long    Capabilities;           /* Capabilities of video card      */
    unsigned far *VideoModePtr;     /* Pointer to supported modes      */
    short   TotalMemory;            /* Number of 64kb memory blocks    */
    char    reserved[236];          /* Pad to 256 byte block size      */
} VbeInfoBlock;

/* SuperVGA mode information block */

struct
{
    unsigned short ModeAttributes;      /* Mode attributes               */
    unsigned char  WinAAttributes;      /* Window A attributes           */
    unsigned char  WinBAttributes;      /* Window B attributes           */
    unsigned short WinGranularity;      /* Window granularity in k       */
    unsigned short WinSize;             /* Window size in k              */
    unsigned short WinASegment;         /* Window A segment              */
    unsigned short WinBSegment;         /* Window B segment              */
    void (far *WinFuncPtr)(void);       /* Pointer to window function    */
    unsigned short BytesPerScanLine;    /* Bytes per scanline            */
    unsigned short XResolution;         /* Horizontal resolution         */
    unsigned short YResolution;         /* Vertical resolution           */
    unsigned char  XCharSize;           /* Character cell width          */
    unsigned char  YCharSize;           /* Character cell height         */
    unsigned char  NumberOfPlanes;      /* Number of memory planes       */
    unsigned char  BitsPerPixel;        /* Bits per pixel                */
    unsigned char  NumberOfBanks;       /* Number of CGA style banks     */
    unsigned char  MemoryModel;         /* Memory model type             */
    unsigned char  BankSize;            /* Size of CGA style banks       */
    unsigned char  NumberOfImagePages;  /* Number of images pages        */
    unsigned char  res1;                /* Reserved                      */
    unsigned char  RedMaskSize;         /* Size of direct color red mask   */
    unsigned char  RedFieldPosition;    /* Bit posn of lsb of red mask     */
    unsigned char  GreenMaskSize;       /* Size of direct color green mask */
    unsigned char  GreenFieldPosition;  /* Bit posn of lsb of green mask   */
    unsigned char  BlueMaskSize;        /* Size of direct color blue mask  */
    unsigned char  BlueFieldPosition;   /* Bit posn of lsb of blue mask    */
    unsigned char  RsvdMaskSize;        /* Size of direct color res mask   */
    unsigned char  RsvdFieldPosition;   /* Bit posn of lsb of res mask     */
    unsigned char  DirectColorModeInfo; /* Direct color mode attributes    */
    unsigned char  res2[216];           /* Pad to 256 byte block size      */
} ModeInfoBlock;

typedef enum
```

```
{
    memPL       = 3,                /* Planar memory model              */
    memPK       = 4,                /* Packed pixel memory model        */
    memRGB      = 6,                /* Direct color RGB memory model    */
    memYUV      = 7,                /* Direct color YUV memory model    */
} memModels;

/*------------------------- Global Variables ---------------------------*/
char mystr[256];
char *get_str();

int     xres,yres;                  /* Resolution of video mode used    */
int     bytesperline;               /* Logical CRT scanline length      */
int     curBank;                    /* Current read/write bank          */
unsigned int bankShift;             /* Bank granularity adjust factor   */
int     oldMode;                    /* Old video mode number            */
char    far *screenPtr;             /* Pointer to start of video memory */
void    (far *bankSwitch)(void);    /* Direct bank switching function   */
/*----------------------- VBE Interface Functions ------------------------*/

/* Get SuperVGA information, returning true if VBE found */

int getVbeInfo()
{
    union REGS in,out;
    struct SREGS segs;
    char far *VbeInfo = (char far *)&VbeInfoBlock;
    in.x.ax = 0x4F00;
    in.x.di = FP_OFF(VbeInfo);
    segs.es = FP_SEG(VbeInfo);
    int86x(0x10, &in, &out, &segs);
    return (out.x.ax == 0x4F);
}

/* Get video mode information given a VBE mode number. We return 0 if
 * if the mode is not available, or if it is not a 256 color packed
 * pixel mode.
 */

int getModeInfo(int mode)
{
    union REGS in,out;
    struct SREGS segs;
    char far *modeInfo = (char far *)&ModeInfoBlock;
    if (mode < 0x100) return 0;     /* Ignore non-VBE modes             */
    in.x.ax = 0x4F01;
    in.x.cx = mode;
    in.x.di = FP_OFF(modeInfo);
    segs.es = FP_SEG(modeInfo);
    int86x(0x10, &in, &out, &segs);
    if (out.x.ax != 0x4F) return 0;
    if ((ModeInfoBlock.ModeAttributes & 0x1)
            && ModeInfoBlock.MemoryModel == memPK
            && ModeInfoBlock.BitsPerPixel == 8
            && ModeInfoBlock.NumberOfPlanes == 1)
        return 1;
    return 0;
}

/* Set a VBE video mode */
```

```
void setVBEMode(int mode)
{
    union REGS in,out;
    in.x.ax = 0x4F02; in.x.bx = mode;
    int86(0x10,&in,&out);
}

/* Return the current VBE video mode */

int getVBEMode(void)
{
    union REGS in,out;
    in.x.ax = 0x4F03;
    int86(0x10,&in,&out);
    return out.x.bx;
}

/* Set new read/write bank. We must set both Window A and Window B, as
 * many VBE's have these set as separately available read and write
 * windows. We also use a simple (but very effective) optimization of
 * checking if the requested bank is currently active.
 */

void setBank(int bank)
{
    union REGS  in,out;
    if (bank == curBank) return;    /* Bank is already active           */
    curBank = bank;                 /* Save current bank number         */
    bank <<= bankShift;             /* Adjust to window granularity     */
#ifdef  DIRECT_BANKING
    setbxdx(0,bank);
    bankSwitch();
    setbxdx(1,bank);
    bankSwitch();
#else
    in.x.ax = 0x4F05; in.x.bx = 0;  in.x.dx = bank;
    int86(0x10, &in, &out);
    in.x.ax = 0x4F05; in.x.bx = 1;  in.x.dx = bank;
    int86(0x10, &in, &out);
#endif
}

/*------------------------- Application Functions -----------------------*/

/* Plot a pixel at location (x,y) in specified color (8 bit modes only) */

void putPixel(int x,int y,int color)
{
    long addr = (long)y * bytesperline + x;
    setBank((int)(addr >> 16));
    *(screenPtr + (addr & 0xFFFF)) = (char)color;
}

/* Draw a line from (x1,y1) to (x2,y2) in specified color */

void line(int x1,int y1,int x2,int y2,int color)
{
    int     d;                      /* Decision variable                */
    int     dx,dy;                  /* Dx and Dy values for the line    */
```

```
    int     Eincr,NEincr;           /* Decision variable increments     */
    int     yincr;                  /* Increment for y values           */
    int     t;                      /* Counters etc.                    */

#define ABS(a)   ((a) >= 0 ? (a) : -(a))

    dx = ABS(x2 - x1);
    dy = ABS(y2 - y1);
    if (dy <= dx)
    {
        /* We have a line with a slope between -1 and 1
         *
         * Ensure that we are always scan converting the line from left to
         * right to ensure that we produce the same line from P1 to P0 as the
         * line from P0 to P1.
         */
        if (x2 < x1)
        {
            t = x2; x2 = x1; x1 = t;    /* Swap X coordinates           */
            t = y2; y2 = y1; y1 = t;    /* Swap Y coordinates           */
        }
        if (y2 > y1)
            yincr = 1;
        else
            yincr = -1;
        d = 2*dy - dx;              /* Initial decision variable value  */
        Eincr = 2*dy;               /* Increment to move to E pixel     */
        NEincr = 2*(dy - dx);       /* Increment to move to NE pixel    */
        putPixel(x1,y1,color);      /* Draw the first point at (x1,y1)  */

        /* Incrementally determine the positions of the remaining pixels */
        for (x1++; x1 <= x2; x1++)
        {
            if (d < 0)
                d += Eincr;         /* Choose the Eastern Pixel         */
            else
            {
                d += NEincr;        /* Choose the North Eastern Pixel   */
                y1 += yincr;        /* (or SE pixel for dx/dy < 0!)     */
            }
            putPixel(x1,y1,color);  /* Draw the point                   */
        }
    }
    else
    {
        /* We have a line with a slope between -1 and 1 (ie: includes
         * vertical lines). We must swap our x and y coordinates for this.
         *
         * Ensure that we are always scan converting the line from left to
         * right to ensure that we produce the same line from P1 to P0 as the
         * line from P0 to P1.
         */
        if (y2 < y1)
        {
            t = x2; x2 = x1; x1 = t;    /* Swap X coordinates           */
            t = y2; y2 = y1; y1 = t;    /* Swap Y coordinates           */
        }
        if (x2 > x1)
            yincr = 1;
        else
```

```
            yincr = -1;
        d = 2*dx - dy;              /* Initial decision variable value  */
        Eincr = 2*dx;               /* Increment to move to E pixel     */
        NEincr = 2*(dx - dy);       /* Increment to move to NE pixel    */
        putPixel(x1,y1,color);      /* Draw the first point at (x1,y1)  */

        /* Incrementally determine the positions of the remaining pixels */
        for (y1++; y1 <= y2; y1++)
        {
            if (d < 0)
                d += Eincr;         /* Choose the Eastern Pixel         */
            else
            {
                d += NEincr;        /* Choose the North Eastern Pixel   */
                x1 += yincr;        /* (or SE pixel for dx/dy < 0!)     */
            }
            putPixel(x1,y1,color);  /* Draw the point                   */
        }
    }
}
```

```
/* Draw a simple moire pattern of lines on the display */
void drawMoire(void)
{
    int     i;

    for (i = 0; i < xres; i += 5)
    {
        line(xres/2,yres/2,i,0,i % 0xFF);
        line(xres/2,yres/2,i,yres,(i+1) % 0xFF);
    }
    for (i = 0; i < yres; i += 5)
    {
        line(xres/2,yres/2,0,i,(i+2) % 0xFF);
        line(xres/2,yres/2,xres,i,(i+3) % 0xFF);
    }
    line(0,0,xres-1,0,15);
    line(0,0,0,yres-1,15);
    line(xres-1,0,xres-1,yres-1,15);
    line(0,yres-1,xres-1,yres-1,15);
}

/* Return NEAR pointer to FAR string pointer*/

char *get_str(char far *p)
{
    int i;
    char *q=mystr;

    for(i=0;i<255;i++)
    {
       if(*p) *q++ = *p++;
       else break;
    }
    *q = '\0';
    return(mystr);
}

/* Display a list of available resolutions. Be careful with calls to
 * function 00h to get SuperVGA mode information. Many VBE's build the
 * list of video modes directly in this information block, so if you
 * are using a common buffer (which we aren't here, but in protected
 * mode you will), then you will need to make a local copy of this list
 * of available modes.
 */

void availableModes(void)
{
    unsigned far    *p;

    if (!getVbeInfo())
    {
        printf("No VESA VBE detected\n");
        exit(1);
    }
    printf("VESA VBE Version %d.%d detected (%s)\n\n",
        VbeInfoBlock.VESAVersion >> 8, VbeInfoBlock.VESAVersion & 0xF,
        get_str(VbeInfoBlock.OEMStringPtr));
    printf("Available 256 color video modes:\n");
    for (p = VbeInfoBlock.VideoModePtr; *p !=(unsigned)-1; p++)
```

```
    {
        if (getModeInfo(*p))
        {
            printf("    %4d x %4d %d bits per pixel\n",
                ModeInfoBlock.XResolution, ModeInfoBlock.YResolution,
                ModeInfoBlock.BitsPerPixel);
        }
    }
    printf("\nUsage: hellovbe <xres> <yres>\n");
    exit(1);
}

/* Initialize the specified video mode. Notice how we determine a shift
 * factor for adjusting the Window granularity for bank switching. This
 * is much faster than doing it with a multiply (especially with direct
 * banking enabled).
 */
void initGraphics(unsigned int x, unsigned int y)
{
    unsigned far    *p;

    if (!getVbeInfo())
    {
        printf("No VESA VBE detected\n");
        exit(1);
    }
    for (p = VbeInfoBlock.VideoModePtr; *p != (unsigned)-1; p++)
    {
        if (getModeInfo(*p) && ModeInfoBlock.XResolution == x
                && ModeInfoBlock.YResolution == y)
        {
           xres = x;   yres = y;
           bytesperline = ModeInfoBlock.BytesPerScanLine;
           bankShift = 0;
           while ((unsigned)(64 >> bankShift) != ModeInfoBlock.WinGranularity)
               bankShift++;
           bankSwitch = ModeInfoBlock.WinFuncPtr;
           curBank = -1;
           screenPtr = (char far *)( ((long)0xA000)<<16 | 0);
           oldMode = getVBEMode();
           setVBEMode(*p);
           return;
        }
    }
    printf("Valid video mode not found\n");
    exit(1);
}

/* Main routine. Expects the x & y resolution of the desired video mode
 * to be passed on the command line. Will print out a list of available
 * video modes if no command line is present.
 */
void main(int argc,char *argv[])
{
    int x,y;

    if (argc != 3)
        availableModes();       /* Display list of available modes      */

    x = atoi(argv[1]);          /* Get requested resolution             */
```

```
    y = atoi(argv[2]);
    initGraphics(x,y);          /* Start requested video mode          */
    drawMoire();                /* Draw a moire pattern                */
    getch();                    /* Wait for keypress                   */
    setVBEMode(oldMode);        /* Restore previous mode               */
}

/*--------------------------------------------------------------------*/
/* The following commented-out routines are for Planar modes          */
/* outpw() is for word output, outp() is for byte output              */
/*--------------------------------------------------------------------*/

/* Initialize Planar (Write mode 2)
 * Should be Called from initGraphics

void initPlanar()
{
   outpw(0x3C4,0x0F02);
   outpw(0x3CE,0x0003);
   outpw(0x3CE,0x0205);
}
*/

/* Reset to Write Mode 0
 * for BIOS default draw text

void setWriteMode0()
{
   outpw(0x3CE,0xFF08);
   outpw(0x3CE,0x0005);
}
*/

/* Plot a pixel in Planar mode

void putPixelP(int x, int y, int color)
{
    char dummy_read;

    long addr = (long)y * bytesperline + (x/8);
    setBank((int)(addr >> 16));
    outp(0x3CE,8);
    outp(0x3CF,0x80 >> (x & 7));
    dummy_read = *(screenPtr + (addr & 0xFFFF));
    *(screenPtr + (addr & 0xFFFF)) = (char)color;
}
*/
```

Assembly Language Module

Below is the Assembly Language module required for the direct bank switching. In Borland C or other C compilers, this can be converted to in-line assembly code.

```
public _setbxdx
.MODEL SMALL            ;whatever
.CODE
set_struc       struc
        dw      ?       ;old bp
        dd      ?       ;return addr (always far call)
```

```
p_bx    dw      ?       ;reg bx value
p_dx    dw      ?       ;reg dx value
set_struc       ends

_setbxdx        proc far     ; must be FAR
        push    bp
        mov     bp,sp
        mov     bx,[bp]+p_bx
        mov     dx,[bp]+p_dx
        pop     bp
        ret
_setbxdx        endp
END
```

Appendix 3 - Differences Between VBE Revisions

VBE 1.0

Initial implementation: Implemented Functions 00-05h
Defined modes 100-107h

VBE 1.1

Second implementation: Added Functions 06h and 07h.
Added modes 108-10Ch
Added TotalMemory to VbeInfoBlock
Added NumberOfImagePages and
Reserved fields to ModeInfoBlock

VBE 1.2

Third implementation: Added Function 08h
Added Hi-color modes 10D-11Bh
Added Reserved field to VbeInfoBlock
Added New Direct color fields to ModeInfoBlock
Changed optional fields to mandatory in ModeInfoBlock
Added Capabilities bit definition in VbeInfoBlock

VBE 2.0

Fourth implementation: Added Flat Frame Buffer support in Function 02h (D14)
Added protected mode support (Function 0Ah)
Added new DAC services for palette operations
(Function 09h)
Added new completion codes 02h and 03h
Added OEM information to VbeInfoBlock
Added two new definitions to Capabilities in VbeInfoBlock
Added new fields to ModeInfoBlock
Certification and ROM requirements for Compliance
Clarified Memory Clear bit in Function 02h (D15)
Clarified Memory Clear bit in Function 03h (D15)
Added new return field in Function 06h
Added Supplemental Functions definition and defined
Supplemental Functions 10-16h
Added new mode to access all of video memory
Added wait for vertical retrace in Function 07h
Clarified and removed ambiguities in the earlier

specifications
Added new mode to access all video memory.

VBE 2.0, Rev. 1.1

Fifth implementation:

Section 3 - Revised sentence to read: Note that modes may only be set if the mode exists in the VideoModeList pointed to by the VideoModePTR returned in Function 00h.
The exception to this requirement is the mode number 81ffh.

Section 4.2 - Added: If the memory location is zero, then only I/O mapped ports will be used so the application does not need to do anything special. This should be the default case for ALL cards that have I/O mapped registers because it provides the best performance.
and
If the memory location is nonzero (there can be only one), the application will need to create a new 32-bit selector with the base address that points to the "physical" location specified with the specified limit.
and
When the application needs to call the 32-bit bank switch function, it must then load the ES selector with the value of the new selector that has been created. The bank switching code can then directly access its memory mapped registers as absolute offsets into the ES selector
(i.e., mov [es:10],eax to put a value into the register at base+10).

It is up to the application code to save and restore the previous state of the ES selector if this is necessary (for example in flat model code)

Section 4.5 - Revised sentence to read: If function call D7 is set and the application assumes it is similar to the IBM compatible mode set using VBE Function 02h, the implementation will fail.

Added: **Note:** CX and DX, for both input and output values, will be zero based.

Added: If the memory location is zero, then only I/O mapped ports will be used so the application does not need to do anything special. This should be the default case for

ALL cards that have I/O mapped registers because it provides the best performance.
and
If the memory location is nonzero (there can be only one), the application will need to create a new 32-bit selector with the base address that points to the “physical” location specified with the specified limit.
and
When the application needs to call the 32-bit bank switch function, it must then load the ES selector with the value of the new selector that has been created. The bank switching code can then directly access its memory mapped registers as absolute offsets into the ES selector (i.e., mov [es:10],eax to put a value into the register at base+10).

It is up to the application code to save and restore the previous state of the ES selector if this is necessary (for example in flat model code).

Added to first paragraph: However, it should not appear in the VideoModeList. Look in the ModeInfoBlock to determine if paging is required and, if it is required, how it is supported.

Section A5.2.11 - Added: If the memory location is zero, then only I/O mapped ports will be used so the application does not need to do anything special. This should be the default case for ALL cards that have I/O mapped registers because it provides the best performance.
and
If the memory location is nonzero (there can be only one), the application will need to create a new 32-bit selector with the base address that points to the “physical” location specified with the specified limit.

Corrected typographical errors and style.

Modified copyright notice; modified Support section; added missing paragraphs regarding protected mode to function 0Ah and section on protected mode considerations; corrected typo in function 09h – 255 should have read 256; corrected cast of 'color' in C example.

VBE 3.0

Current implementation:

Major style and formatting updated to the document to enhance the visual appearance and readability.

Added new ‘Programming with VBE/Core’ section for application programming notes.

Added new ‘Protected Mode Entry Point’ for compatibility with fully protected mode operating systems such as Windows NT, OS/2 and the many variants of UNIX.

VBE Function 00h. Updated to include support for refresh control.

VBE Function 00h. Added note about extended text modes not required to be supported via the protected mode entry point.

VBE Function 01h. Updated to return extended information about the banked and linear modes, so that the hardware implementation for banked and linear modes can be different (such as restricted memory access in one of the modes, different scanline widths etc).

VBE Function 01h. Updated to return information about the maximum pixel clock possible for the graphics mode, along with the pixel clock scaling flags.

VBE Function 02h. Updated to provide a ‘Refresh Control’ flag in bit D13 of the video mode number, and to pass in a table of normalized CRTC values and pixel clock to the mode set function.

VBE Function 02h. Changed so that VBE 3.0 implementations may not clear the screen for a mode set if bit D15 is set. VBE 3.0 aware apps must assume screen has not been cleared.

VBE Function 02h. Added note about extended text modes not required to be supported via the protected mode entry point.

VBE Function 07h. Updated to provide support for hardware triple buffering and Stereo LC shutter glasses.

VBE Function 09h. Updated to be optional if the controller is VGA compatible to save on BIOS space.

VBE Function 0Ah. Updated to be optional since the new protected mode entry point supercedes this interface.

Appendix 4 - Related Documents

- VGA Reference Manual(s)
- Graphic Controller Data Sheets
- VESA Monitor Timings
- VBE/PM Monitor Power Management Standard
- VBE/AI VESA Audio Interface
- VBE/AF Accelerator Functions Specification
- VBE/DDC VESA Display Data Channel Software Interface Standard
- VESA DDC Hardware Specification
- VESA DPMS Hardware Specification
- VESA Generalized Timing Formula (VM/GTF) Specification
- VESA Stereo Connector (Proposal)