VBE/AF Standard

Video Electronics Standards Association

860 Hillview Court, Suite 150
Milpitas, CA 95035

Phone: (408) 957-9270
FAX: (408) 957-9277

VESA BIOS Extension/Accelerator Functions (VBE/AF)

Version: 1.0
Revision: 0.7
Adoption Date: August 18, 1996

Purpose

To define a standard method for software and drivers to access the accelerator features present in most graphics hardware.

Summary

This document defines the interface for a new operating system portable, loadable device driver architecture that will provide access to accelerated graphics hardware. Some of the accelerator functions supported include hardware cursors, multi buffering, solid and transparent off-screen bitmaps, rectangle filling, line drawing and polygon filling. This specification is not just targeted at supporting graphics accelerator hardware, but can also be used to support existing dumb framebuffer devices with maximum performance.

VBE/AF is essentially a hardware-independent device interface at the very lowest level. It is not meant to replace higher-level APIs, but it will allow those APIs and applications that want direct hardware support to access the hardware in a standardized way.

Intellectual Property

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.

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.

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 questions must be writing to VESA, in order of preference at:

Internet: support@vesa.org

FAX: (408) 957-9277

MAIL: VESA
860 Hillview Court, Suite 150
Milpitas, CA 95035

SSC VBE/AF Workgroup Members

Any industry standard requires input from many sources.

The people listed below were members of the VBE/AF Workgroup of the Software Standards Committee (SSC) which was responsible for combining all of the industry input into this standard are listed below.

The workgroup members would also like to express a special gratitude to Kendall Bennett for his significant contributions to making VBE/AF a reality.

VBE/AF Workgroup

David Penley, Cirrus Logic, Inc. - Workgroup Chairman

Kendall Bennett, SciTech Software
Kevin Gillett, S-MOS Systems, Inc.,
Brad Haakenson, Cirrus Logic, Inc.
Jake Richter, Panacea Inc.
Tom Ryan, SciTech Software

Revision History:

0.1: Initial revision put forward for proposal
0.2: Formatting and editing changes for distribution
0.3: More formatting and editing changes for distribution. Rewrite some of the Introduction
0.4: Still more formatting and editing changes. Added cross-references to many parts.
0.5: Added missing BIOS call interface. Fixed up some missing items in header structures.
Updated with latest source code..
0.6: Added support for multi-buffering for games requiring triple and higher buffering.
0.7: Removed support for BIOS call interface, to be moved into an alternative BIOS specification.

Table of Contents

Introduction

This document contains the Graphics Accelerator Driver Architecture specification (hereafter VBE/AF) for standard software access to high performance accelerated graphics display controllers that support resolutions, color depths, and frame buffer organizations beyond the original VGA hardware standard.

To understand the VBE/AF specification, some knowledge of 80386 assembly language and Accelerated SuperVGA hardware may be required.

VBE/AF Features

- Standard application interface to graphical user interface (GUI) accelerator devices.
- Operating system (OS) neutral high performance 32 bit loadable device driver.
- Support for common GUI accelerator functions.
- Support for pixel depths from 8 bit to 32 bits per pixel.
- Support for basic mix operations (REPLACE, OR, AND, XOR).
- Support for offscreen memory management for bitmap storage.
- Support for multi buffering for flicker free animation.
- Support for virtual scrolling for large desktops and arcade style games.

Scope of VBE/AF

The primary purpose of the VBE/AF is to provide standard software support for the many unique implementations of high performance Accelerated SuperVGA (SVGA) graphics controllers on the PC platform. VBE/AF is intended to provide computer game and interactive graphics software developers on the PC standardized access to accelerated display hardware without requiring any OEM specific display drivers.

On the other side, manufacturers can reduce their burden of software support by implementing a single VBE/AF driver for their graphics products. With this one driver, support for games and low volume operating systems can be realized with a minimal effort.

The VBE/AF device driver is defined as an operating system portable, 32 bit protected mode device driver, which can be loaded under any 32 bit operating system. The same VBE/AF driver can be used for high performance full screen graphics under any operating system, provided a few operating system specific functions are provided. Currently VBE/AF is targeted towards supporting MSDOS, Windows 3.1, Windows 95, OS/2 and UNIX.

VBE/AF is similar in functionality to the VESA VBE Core Functions 2.0 specification, but because new and future accelerated device's are so vastly different to the generic VGA and

SuperVGA devices originally supported by VBE 2.0, VBE/AF provides a completely new device driver interface.

VBE/AF provides access to a standard set of accelerator functions provided by the display controller, and reports the availability and details of all supported functions to the application as necessary. VBE/AF also provides information on how to set up and maintain the state that the VBE/AF device driver needs for correct operation (such as accessing I/O and memory mapped hardware registers). If a hardware device does not support a particular accelerator function, such as line drawing, support for this function will be mapped out and the application will need to implement it in software. Although there is generally a 1:1 mapping between VBE/AF functions and accelerator hardware functions, some VBE/AF functions (such as polygon filling) will need to be supported with a software rasterizer if not supported directly in hardware. This is done specifically for the sake of efficiency, and the fact that many hardware devices support varying levels of polygon rendering support.

VBE/AF only provides support for 8 bit and higher packed pixel video modes. Although VBE/AF is intended to support accelerated hardware, it also provides support for dumb framebuffer video modes without any hardware acceleration. This is provided for backwards compatibility with older non-accelerated hardware, and to support video modes that cannot be accelerated by the underlying hardware.

VBE/AF is also designed to correctly arbitrate between accelerated device access and dumb framebuffer access for the installed display device. This allows application code to take advantage of accelerator functions in certain areas, or perform direct software rendering into banked or linear framebuffer video memory. VBE/AF compatible devices must support at least either a banked framebuffer direct access mode, or a linear framebuffer access mode.

In order for the VBE/AF specification to be acceptable and used by application developers, it must be fast and flexible. Application developers must be able to use the functions to implement high performance, complex accelerated graphics primitives by building upon the accelerated building blocks provided by the VBE/AF. For this reason, the VBE/AF functions have been boiled down into the smallest possible set of accelerator functions that are generally implemented directly in hardware. When an application calls the VBE/AF device driver routines, it will already have performed any necessary application specific special casing. This means that the VBE/AF device driver routines will be short and will simply program the hardware and return without doing any parameter checking, clipping etc.

Background

Since the introduction of Microsoft Windows 3.0, PC based graphics hardware has quickly developed from the original IBM VGA standard to incredibly high performance hardware that supports pixel resolutions and depths way beyond the original IBM VGA standard. In order to be able to handle the massive bandwidth that Graphical User Interface (or GUI) operating systems like Windows require in these extended resolutions, these devices have also incorporated a number of extensions for off-loading the burden of graphics rendering from the CPU to the graphics hardware device. This advanced hardware acceleration (or GUI acceleration) has now become the defacto industry standard, and any new PC based graphics devices developed today at least contain some sort of GUI hardware acceleration.

However, several serious problems face a software developer who intends to take advantage of these "GUI Accelerator" devices. Because there is no standard hardware implementation, the developer is faced with widely disparate hardware accelerator architectures. Lacking a common software interface, designing applications for these environments is costly and technically difficult. Except for applications or Operating Systems supported by OEM-specific display drivers, very few software packages can take advantage of the power and capabilities of GUI hardware accelerator products.

VBE/AF is intended to bridge that gap, allowing computer games and other interactive graphics software to have a standardized, fast and flexible interface to high performance GUI hardware accelerator functions. This allows application developers to spend more time developing the core of the application, than writing hardware specific device drivers.

Programming with VBE/AF

This chapter describes in detail the issues application and system software developers will face when developing code to use VBE/AF.

Initializing the Device Context Interface

In order to be able to use the VBE/AF device driver, it must be dynamically loaded and initialized by the application or operating system driver. The first step is to locate the VBE/AF device driver file in the operating system's normal file space, or to read the VBE/AF device driver file from non-volatile memory from the graphics card (please contact VESA for more information on the forthcoming VESA standard to cover this). The location of this file is operating system dependent, and will be found in the following standard locations:

Operating System	*Default Location*
MSDOS	C:\VBEAF.DRV
Windows '95	C:\VBEAF.DRV
Windows NT	C:\VBEAF.DRV
OS/2	C:\VBEAF.DRV
UNIX	/VBEAF.DRV

Note however that the user may also set the VBEAF_PATH environment variable and the loader library should also check in this location if the driver file cannot be found in the default location.

VESA is currently defining an alternative function to read the VBE/AF device driver block directly from non-volatile storage from the graphics controller. This function is optional and may not be present, so you must first check the availability of this function before you can use it (refer to the forthcoming VESA spec for more info). Note also that this function should be used as a last resort. If a valid driver file is found in the operating systems normal file space, this driver file must be used instead to ensure that the VBE/AF driver file can be upgraded in the field.

Once the device driver file has been located, it will need to be loaded into the 32 bit linear address space of the application or operating system driver. The application should allocate enough space to hold the entire device driver file with a standard operating system memory allocation request, and then load the device driver file into this memory block using standard operating system file services.

Once the device driver file is loaded, the initialization function must be called. When the device driver is first loaded from disk, the *InitDriver* entry in the device driver function table will contain an offset from the start of the driver to the initialization function. The application must then make a near call to the address where the initialization function is located using this offset.

Before the *InitDriver* function can be called however, the application must map the necessary physical memory location into the applications linear address space and store the mapped locations in the device context buffer, allocate the special selectors to physical memory locations in the first 1Mb of system memory and provide the necessary IO permissions required by the device driver (the easiest is simply to provide IOPL for the code). The application must also fill in the *Int86* and *CallRealMode* device callback functions to point to operating system specific functions to provide these services. These callbacks are necessary for the VBE/AF driver to be able to communicate with the real mode Video BIOS stored on the graphics card in order to initialize video modes and get video card information.

When the VBE/AF device driver functions are called, the current I/O permission bit map must allow access to the I/O ports that the VBE/AF functions may need to access (or the process must be running with full I/O privileges). The VBE/AF device driver may also need to directly access memory mapped registers located in *physical* extended memory, and directly access the banked and linear framebuffer memory locations. Access to these *physical* memory locations is provided by the application by mapping the memory locations that the device may need into the near address space used when the VBE/AF functions are called (such as using DPMI function 0x800). The actual linear address of the mapping is stored in the accelerator device context buffers *IOMemMaps*, *BankedMem* and *LinearMem* variables, and will be used as necessary by the device driver. Note that the base address of these locations is specified as a *physical* memory address, and may be located anywhere in physical memory (including well above the 1Mb memory mark), so an operating specific service will need to be used to map this memory location into the processes linear address space. As well a providing near pointer access to memory locations, the application must also allocate a number of special selectors to access common areas of real mode memory (in the same task that is used to implement the *Int86* and *CallRealMode* callbacks), such as the BIOS data area at 0x40:0 and the Video BIOS segment at 0xC000:0. These selectors are provided for the convenience of the developer of the VBE/AF implementation, and allows a single source to be used to build both 16 bit BIOS initialization code and 32 bit VBE/AF initialization code.

Note that in order to be able to call the device driver code that has been loaded on the heap, the application must have a code selector that points to the same linear address space as the default DS selector. Under most Intel 386 flat model architectures the default process CS and DS selectors point to the same linear address space, so you simply need to make a near call within the current CS selector (providing the pages that the driver has been loaded into have executable access). If this is not the case (perhaps in an OS device driver) the appropriate selectors or memory mappings will need to be created and loaded as necessary.

When the application calls the *InitDriver* function, it will perform the following functions, and will return the appropriate status code in the EAX register (see InitDriver function description on page 19 for more details):

- Verify the presence of the supported graphics adapter
- Load the near addresses of initialization functions in device context

Note that because the device driver will have been pre-configured by the external configuration program *before* the application loads it, all of the device driver configuration information such as

the list of available video modes, available memory etc. will be correctly set. The initialization function will however verify the presence of the expected graphics controller, and will fail to initialization if the graphics controller has been changed.

After initializing the device driver, the application can then simply make near calls to the initialization functions located in the device driver to obtain information about available video modes and to initialize a desired video mode. Note that none of the addresses for the acceleration functions will be valid until *after* a video mode has been initialized. The VBE/AF driver will determine which functions are available in each video mode, and will load the addresses of the functions appropriate for the selected video mode into the device context function address table. Once the mode has been initialized, the application can simply call any of the supported functions directly.

When calling the functions directly, the stack should also be a valid 32 bit protected mode stack, and should be large enough to allow the accelerator functions to use up to 1024 bytes of temporary stack space if necessary. The values of EAX, EBX, ECX, EDX, ESI and EDI are determined by the appropriate device context function being called, and the value of all other registers will be preserved across all calls to the VBE/AF device driver.

Also all VBE/AF device driver functions expect the address of the loaded device driver context to be passed in the DS:EBX registers. Since in 32 bit protected mode code, the DS register is always set to the applications default data segment, the application simply needs to load the address of the device context buffer into EBX before calling the VBE/AF device driver functions and will never need to load/restore any selectors.

Initializing a Desired Video Mode

Before any of the accelerator functions can be called, one of the supported video modes must be initialized by the application program by calling the *SetVideoMode* function. In order to find a valid video mode number to be passed to *SetVideoMode*, the *GetVideoModeInfo* function is used to obtain specific information about all of the available video modes supported by the loaded driver.

VBE/AF does not define any standard video mode numbers to be used for identifying available video modes, but relies on the application to search through the list of available video modes for one that has the desired resolution and pixel depth. Once the desired video mode has been identified, this video mode number can be used in the call to *SetVideoMode*.

Determining Specific Hardware Support

Once a particular video mode has been initialized, the application can make calls to all the available hardware acceleration functions to perform drawing operations. However not all functions may be supported in hardware, and the application will need to determine which functions are supported. The VBE/AF driver will load a NULL (0) into the device driver functions address table locations for all functions that are not supported. For all unsupported functions, the application will need to provide support with it's own software rendering routines

(which may call other supported acceleration functions). The VBE/AF driver will never simulate an optional hardware function in software if it is not available.

Hence on a device that does not support hardware line drawing, after setting the desired video mode, the *DrawLine* function in the VBE/AF device driver address table will have a value of 0 stored in it. It is then up to the software application to perform line drawing in software directly to the framebuffer.

2D Coordinate System

All the VBE/AF accelerator functions take coordinates in a global framebuffer coordinate system. This coordinate system starts with (0,0) at the start of framebuffer memory and increments the X coordinate for every pixel and Y coordinate for every scanline. For instance, if the logical scanline width is set to 1024 bytes, then the coordinate (0,1) will be rendering into the byte at location 1024 from the start of framebuffer memory.

It is up to the application to impose any other logical coordinate system on top of the VBE/AF device driver routines, such as handling viewport mapping etc.

Multi Buffering

Multi buffering is selected by setting the multi buffer bit when initializing the desired video mode (check that multi buffering is supported by the video mode first!). When multi buffering is active, VBE/AF will divide the framebuffer memory into multiple buffers (either logically or in hardware). You can also enable both multi buffering and virtual scrolling at the same time (this may fail on some controllers however), in which case you would have enabled a virtual buffer size during the mode set, and must have enough video memory to accommodate the number of buffers of the specified virtual size (see Virtual Buffer Scrolling on page 9 for more information on this). Access to the currently active buffer is transparent to the calling application when it calls the VBE/AF accelerator functions (all rendering functions automatically offset drawing into the currently active buffer). However if the application is rendering directly to the framebuffer, the application program must use the value of *OriginOffset* to find the starting location of the currently active framebuffer. The value in *OriginOffset* is the linear offset of the active framebuffer from the start of video memory, and is always correctly maintained by VBE/AF.

In the examples below we examine the case of multi buffering using two buffers, which is usually called double buffering. If the offscreen memory resources are available, multi buffing can be more useful because it allows applications to draw continuously without waiting for the vertical retrace when swapping the currently active visible buffer. Because the VBE/AF driver must allocate offscreen video memory for the multiple display buffers, you must specify the number of display buffers that you want when you set a multi buffered video mode. The VBE/AF driver will then allocate the remainder of the video memory as the offscreen buffer for storing bitmaps.

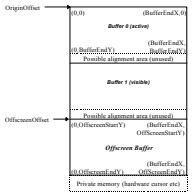

Figure 1: Layout of display memory after SetActiveBuffer(0) and SetVisibleBuffer(1)

All graphics output is sent to the currently active buffer, and all video data is displayed from the currently visible buffer. The value of *BufferEndX* and *BufferEndY* specify the maximum coordinates for the two buffers, and normally will be equal to *Xresolution-1* and *Yresolution-1*. However if virtual scrolling is enabled, these values will be equal to the specified virtual buffer size that was initialized during the mode set, and will be larger than the physical video mode resolution. Double buffering is achieved by calling *SetActiveBuffer* and *SetVisibleBuffer* to set the active and visual buffers to different values, which will cause primitives to be drawn to the hidden buffer, while the CRT controller displays data from a static image in the visible buffer. In the example above, the active buffer is set to buffer 0, while the visible buffer is set to buffer 1. Note however that this is only a conceptual view; the actual hardware may implement double buffering using totally separate framebuffers, in which case the OffscreenBuffer areas will be different for buffers 0 and 1 (the *afHaveDualBuffers* flag will be set for the video mode in the mode attributes field in this case). The visible image can then be instantly updated by swapping the new visible buffer to the buffer that was currently being rendered into.

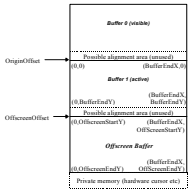

Figure 2: Layout of display memory after SetActiveBuffer(1) and SetVisibleBuffer(0)

Accessing Offscreen Memory

Offscreen video memory on the controller can be used for caching bitmap information to be used for fast BitBlt and transparent BitBlt operations for building up the frames in an animation. To let the application know where the current offscreen memory buffer is located, VBE/AF maintains the *OffscreenOffset, OffscreenStartY* and *OffScreenEndY* state variables, which will always point to the current active offscreen memory areas that the application can use. If the offscreen memory buffer is not available to the application, such as would be the case for lack of video memory, the *OffscreenOffset* state variable will be set to 0.

Note that the values in these variables may change when a call is made to update the currently active rendering buffer when performing multi buffered animation, so the application must always use the current values to determine the location of the offscreen buffer. The actual buffer will still be located in the same physical location in video memory, however it's logical coordinate location may possibly change depending on how the VBE/AF driver implements

multi buffering in hardware (see Figure 1 and Figure 2 to see how this is implemented for most controllers).

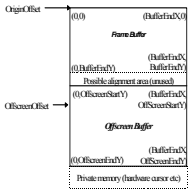

Figure 3: Layout of display memory without multi buffering

Note also that on some devices where double buffering is implemented in hardware as two distinct framebuffer areas, the offscreen memory accessible when the first buffer is active will be different to the offscreen memory accessible when the second buffer is active. Essentially the two hardware framebuffers will be divided up the same as in Figure 3, with one for each hardware framebuffer.

If the VBE/AF application wishes to render into the offscreen memory area directly, the *OffscreenOffset* state variable will always point to the current linear offset of the offscreen memory area from the start of the framebuffer. Note also that VBE/AF provides support for non-conforming BitBlt operations from offscreen video memory, allowing sprites to be stored contiguously in video memory for controllers that support this (as opposed to dividing up the offscreen memory into rectangular regions). The length of the offscreen memory buffer in bytes can be determined by multiplying the number of scan lines in the offscreen buffer (OffscreenEndY - OffscreenStartY + 1) by the logical scanline width for the current video mode. Memory past the end of the offscreen buffer *must not* be used by the application, and is reserved for use by the controller for storing hardware cursors or other internal functions.

Virtual Buffer Scrolling

Virtual scrolling functionality is selected by setting the virtual scroll bit when initializing the desired video mode (check that virtual scrolling is supported by the video mode first!), and specifying a desired virtual framebuffer resolution to be enabled. If there is not enough video memory for the mode (you need twice as much if you enable double buffering as well) then the mode set will fail. Once the video mode is initialized, the application can scroll around within the virtual buffer using the *SetDisplayStart* function to place the display start address at the desired pixel location.

Note that you can enable both virtual scrolling and double buffering at the same time, but on some controllers this may fail (generally if the mode attributes have both the double buffering and virtual scrolling flags enabled, then this should be available. However this is not guaranteed so the application should check that the mode did set correctly). If you enable multi buffering and virtual scrolling at the same time, the coordinates passed for the display start address are logical coordinates within the currently visible buffer. If you call *SetVisibleBuffer*, this will change the logical offset of the first visible pixel in the buffer. If you wish to change the display start address and swap buffers at the same time, then you should call the *SetDisplayStart* function with the *waitVRT* flag set to -1, and then call the *SetVisibleBuffer* routine to re-program

the CRT controller starting address. The call to *SetDisplayStart* will not actually update the CRT controller start address, but will simply update the internal state to reflect the new display starting pixel coordinates that will be used by the subsequent *SetVisibleBuffer* function.

Palette Programming During Double Buffering

If you wish to re-program the palette at the same time as performing double buffering (if the palette changes between frames during double buffering) then you should call the *SetVisibleBuffer* function first with the wait for retrace flag set, then *immediately* following you should program as many palette values as you can before the end of the retrace period. Note that on many RAMDAC devices, you cannot program an entire set of 256 color table entries during the retrace period before the onset of snow, so you will need to stagger the palette change over 2 or more frames. Generally most devices can handle about 100-120 palette entries to be programmed per retrace (many new RAMDAC's don't produce snow at all, so you can program all 256 at once).

Color Values

All color values passed to the accelerated rendering functions are packed pixel values, that will need to be pre-packed into the proper format required by the current framebuffer mode. In 8 bit color index modes, this is simply a color index between 0 and 255. In the 15 bits per pixel and above video modes, you will need to pack the color values according to the RGB pixel format information stored in the mode information block for the current video mode. The RGB pixel format information specifies the mask size and bit positions for all red, green, blue and reserved components. These mask should be used by the application to pack the appropriate RGB color values into a 32 bit integer to be passed to the appropriate rendering routines.

Currently the reserved component in 15 bit and 32 bits per pixel modes is unused, and should ***always*** be set to 0, as on some controllers these bits may be significant.

Integer Coordinates

Integer coordinates are passed as 32 bit signed integers to the accelerated rendering functions. Although the hardware may provide hardware clipping of coordinates outside the normal framebuffer region, the application should ensure that all coordinates passed to the accelerated rendering functions are within the range (0, 0) to (BufferEndX, OffscreenEndY). VBE/AF will always ensure that these values fall within the valid range of values that the hardware is capable of handling.

Fixed Point Coordinates

Fixed point coordinates are passed as 32 bit signed integers in 16.16 fixed point format, where the top 16 bits represent the signed integer portion of the number, and the bottom 16 bits represent the fraction portion of the number. Once again all coordinates passed to rendering functions must be restricted to the range (0, 0) to (BufferEndX, OffscreenEndY).

Fixed point coordinates are used to represent coordinate information for the line drawing, triangle filling and quadrilateral filling routines. This allows the application to specify

coordinates with sub-pixel accuracy, which allows software clipped primitives to rasterize the same set of points that the unclipped primitive would also have rendered.

2D Polygon Rendering

VBE/AF provides routines for high performance polygon rendering functions, which includes flat topped/bottomed trapezoid filling, triangle filling and quadrilateral filling routines. In the cases where the hardware has proper triangle/quad filling support, the triangle and quad functions must be implemented directly. Otherwise the VBE/AF driver will leave these entries marked as not available, but *must* implement the *DrawTrap* function. The trapezoid drawing function will rasterize flat topped and bottomed trapezoids as fast as possible in software using the hardware scanline filling or rectangle filling (or whatever is fastest) routine. The DrawTrap function should not be implemented on devices that have hardware triangle or quad functions.

Note that hardware polygon filling does *not* include devices that rasterize polygons into an offscreen memory area and then perform a BitBlt operation (like 8514/A compatible devices), as the offscreen memory region will not be available for general use by the driver (the application will probably be caching bitmaps in the offscreen memory). These devices should only implement the *DrawTrap* function as fast as possible.

Viewport Transformations

In order to support high performance rendering, but also to allow for fast viewport transformations, the triangle and quad filling functions take both x and y coordinate offset values that are added to all vertices before being processed by the hardware. This allows the hardware rendering functions to implement the translation of coordinates as efficiently as possible (either in hardware or software), and does not require the application passed vertices to be modified in any manner. This is important to ensure that no vertex re-packing is required between sending coordinates from the application program to the hardware for maximum performance. Note also that the *DrawTrap* function does not take x and y coordinate offsets, because it is intended to be used as the back end for a high performance software polygon rasterizing engine.

Direct Framebuffer Access

In order to allow both direct framebuffer access and hardware accelerator access, contention for video memory between the application program and the hardware accelerator must be handled properly for some devices. This is provided by the *EnableDirectAccess* and *DisableDirectAccess* functions. If the *EnableDirectAccess* entry is not NULL in the device context block, then you *must* call these functions prior to performing any direct access to the framebuffer via either the banked memory area or the linear framebuffer memory area, and then call *DisableDirectAccess* before calling any other hardware accelerator functions again.

If the *EnableDirectAccess* function is NULL, you can simply call the *WaitTillIdle* function to ensure the hardware has completed rendering the last command before accessing the framebuffer directly. You need not call any other functions to return to accelerated rendering again.

When the video mode is first initialized, the hardware is automatically set up for hardware accelerated rendering, as though *EnableDirectAccess* was called immediately after setting the video mode.

Driver Function Reference

Structure of the Device Driver

When the device driver is loaded, it can be accessed using the following C structure. Note that after the device driver has been loaded and the *InitDriver* function has been called, all pointers in the device context structure will have been 'fixed up' by the device driver to be real near pointers to the code and data structures within the device driver.

```
typedef struct {
    /*-------------------------------------------------------------------*/
    /* Device driver header block                                        */
    /*-------------------------------------------------------------------*/

    char        Signature[12];      /* 'VBEAF.DRV\0' 12 byte signature  */
    AF_uint32   Version;            /* Driver Interface Version (1.0)   */
    AF_uint32   DriverRev;          /* Driver revision number           */
    char        OemVendorName[80];  /* Vendor Name string               */
    char        OemCopyright[80];   /* Vendor Copyright string          */
    AF_int16    *AvailableModes;    /* Offset to supported mode table   */
    AF_uint32   TotalMemory;        /* Amount of memory in Kb detected  */
    AF_uint32   Attributes;         /* Driver attributes                */
    AF_uint32   BankSize;           /* Bank size in Kb (4Kb or 64Kb)    */
    AF_uint32   BankedBasePtr;      /* Physical addr of banked buffer   */
    AF_uint32   LinearSize;         /* Linear buffer size in Kb         */
    AF_uint32   LinearBasePtr;      /* Physical addr of linear buffer   */
    AF_uint32   LinearGranularity;  /* Linear blt granularity in bytes  */
    AF_uint16   *IOPortsTable;      /* Offset of I/O ports table        */
    AF_uint32   IOMemoryBase[4];    /* Base address of I/O memory maps  */
    AF_uint32   IOMemoryLen[4];     /* Length of I/O memory maps        */
    AF_uint32   res1[10];           /* Reserved for future expansion    */

    /*-------------------------------------------------------------------*/
    /* Near pointers mapped by application for driver                    */
    /*-------------------------------------------------------------------*/

    void        *IOMemMaps[4];      /* Pointers to mapped I/O memory    */
    void        *BankedMem;         /* Ptr to mapped banked video mem   */
    void        *LinearMem;         /* Ptr to mapped linear video mem   */

    /*-------------------------------------------------------------------*/
    /* Important selectors allocated by application for driver           */
    /*-------------------------------------------------------------------*/

    AF_uint32   Sel0000h;           /* 1Mb selector to entire first Mb  */
    AF_uint32   Sel0040h;           /* Selector to segment at 0x0040:0  */
    AF_uint32   SelA000h;           /* Selector to segment at 0xA000:0  */
    AF_uint32   SelB000h;           /* Selector to segment at 0xB000:0  */
    AF_uint32   SelC000h;           /* Selector to segment at 0xC000:0  */

    /*-------------------------------------------------------------------*/
    /* Device driver state variables                                     */
    /*-------------------------------------------------------------------*/

    AF_uint32   BufferEndX;         /* Last X coord of each buffer      */
    AF_uint32   BufferEndY;         /* Last Y coord of each buffer      */
    AF_uint32   OriginOffset;       /* Current start of active page     */
    AF_uint32   OffscreenOffset;    /* Start of offscreen memory area   */
    AF_uint32   OffscreenStartY;    /* First Y coord of offscreen mem   */
    AF_uint32   OffscreenEndY;      /* Last Y coord of offscreen mem    */
    AF_uint32   res2[10];           /* Reserved for future expansion    */

    /*-------------------------------------------------------------------*/
    /* Relocateable 32 bit bank switch routine, needed for framebuffer  */
```

```
/* virtualisation under Windows with DVA.386/VFLATD.386. This       */
/* function *MUST* program the bank with IO mapped registers, as    */
/* when the function is called there is no way to provide access to */
/* the devices memory mapped registers (because there is no way to  */
/* for it to gain access to a copy of this AF_devCtx block). For    */
/* devices that only have memory mapped registers, this vector      */
/* *MUST* be NULL indicating that this is not supported. However    */
/* all these devices all have a real linear framebuffer anyway,     */
/* so the virtualisation services will not be needed.               */
/*------------------------------------------------------------------*/

AF_uint32   SetBank32Len;           /* Length of 32 bit code        */
void        *SetBank32;             /* 32 bit relocateable code     */

/*------------------------------------------------------------------*/
/* REQUIRED callback functions provided by application              */
/*------------------------------------------------------------------*/

void        *Int86;                 /* Issue real mode interrupt    */
void        *CallRealMode;          /* Call a real mode function    */

/*------------------------------------------------------------------*/
/* Device driver functions                                          */
/*------------------------------------------------------------------*/

void        *InitDriver;            /* Initialise driver            */
void        *GetVideoModeInfo;      /* Get video mode information   */
void        *SetVideoMode;          /* Set a video mode             */
void        *RestoreTextMode;       /* Restore text mode operation  */
void        *SetBank;               /* Set framebuffer bank         */
void        *SetDisplayStart;       /* Set virtual display start    */
void        *SetActiveBuffer;       /* Set active output buffer     */
void        *SetVisibleBuffer;      /* Set Visible display buffer   */
void        *SetPaletteData;        /* Program palette data         */
void        *SetGammaCorrectData;   /* Program gamma correct'n data */
void        *WaitTillIdle;          /* Wait till engine is idle     */
void        *EnableDirectAccess;    /* Enable direct mem access     */
void        *DisableDirectAccess;   /* Disable direct mem access    */
void        *SetCursor;             /* Download hardware cursor     */
void        *SetCursorPos;          /* Set cursor position          */
void        *SetCursorColor;        /* Set cursor color             */
void        *ShowCursor;            /* Show/hide cursor             */
void        *SetMix;                /* Set ALU mix operations       */
void        *Set8x8MonoPattern;     /* Set 8x8 mono bitmap pattern  */
void        *Set8x8ColorPattern;    /* Set 8x8 color bitmap pattern */
void        *SetLineStipple;        /* Set 16 bit line stipple      */
void        *SetClipRect;           /* Set clipping rectangle       */
void        *DrawScan;              /* Draw a solid scanline        */
void        *DrawPattScan;          /* Draw a patterned scanline    */
void        *DrawColorPattScan;     /* Draw color pattern scanline  */
void        *DrawScanList;          /* Draw list of solid scanlines */
void        *DrawRect;              /* Draw a solid rectangle       */
void        *DrawPattRect;          /* Draw a patterned rectangle   */
void        *DrawColorPattRect;     /* Draw color pattern rectangle */
void        *DrawLine;              /* Draw a solid line            */
void        *DrawStippleLine;       /* Draw a stippled line         */
void        *DrawTrap;              /* Draw a solid trapezoid       */
void        *DrawTri;               /* Draw a solid triangle        */
void        *DrawQuad;              /* Draw a solid quad            */
void        *PutMonoImage;          /* Display a monochrome bitmap  */
void        *BitBlt;                /* Blt screen to screen         */
void        *BitBltLin;             /* Linear source BitBlt         */
void        *TransBlt;              /* Blt scr/scr w/ transparency  */
void        *TransBltLin;           /* Linear source TransBlt       */
} AF_devCtx;
```

The *Signature* field is filled with the null terminated string ‘VBEAF.DRV’ by the VBE/AF implementation. This can be used to verify that the file loaded really is an VBE/AF device driver.

The *Version* field is a BCD value which specifies what revision level of the VBE/AF specification is implemented in the software. The high byte specifies the major version number and the low byte specifies the minor version number. For example, the BCD value for VBE/AF 1.0 is 0x100 and the BCD value for VBE/AF 1.2 would be 0x102.

The *DriverRev* field specifies the driver revision level, and is used by the driver configuration software to determine which version was used to generate the driver file.

The *OemVendorName* field contains the name of the vendor that developed the device driver implementation, and can be up to 80 characters in length.

The *OemCopyright* field contains a copyright string for the vendor that developed the device driver implementation.

The *AvailableModes* is an offset within the loaded driver to a list of mode numbers for all display modes supported by the VBE/AF driver. Each mode number occupies one word (16 bits), and is terminated by a -1 (0FFFFh). Any modes found in this list are guaranteed to be available for the current configuration.

The *TotalMemory* field indicates the maximum amount of memory physically installed and available to the frame buffer in 1Kb units. Note that not all video modes will be able to address all of this memory.

The *Attributes* field contains a number of flags that describes certain important characteristics of the graphics controller. The fields are exactly the same as those provided in the *AF_modeInfo* block for each video mode, but the meaning is slightly different. For each flag defined below, it represents the whether the controller can support these modes in *any* available video modes. Please see the GetVideoModeInfo function on page 19 for a detailed description of each flag's meaning.

```
#define afHaveMultiBuffer   0x0001
#define afHaveVirtualScroll 0x0002
#define afHaveBankedBuffer  0x0004
#define afHaveLinearBuffer  0x0008
#define afHaveAccel2D       0x0010
#define afHaveDualBuffers   0x0020
#define afHaveHWCursor      0x0040
#define afHave8BitDAC       0x0080
```

The *BankSize* field contains the size of the banked memory buffer in 1Kb units. It can be either 4Kb or 64Kb in length.

The *BankedBasePtr* field is a 32 bit physical memory address where the banked framebuffer memory window is located in the CPU address space. If the banked framebuffer mode is not available, then this field will be zero.

The *LinearSize* is a 32 bit length of the linear frame buffer memory in 1Kb units. In can be any length up to the size of the available video memory.

The *LinearBasePtr* is a 32 bit physical address of the start of frame buffer memory when the controller is in linear frame buffer memory mode. If the linear framebuffer is not available, then this field will be zero.

The *LinearGranularity* field specifies the granularity of the source linear offset to be used for the *BitBltLin* and *TransBltLin* functions. Generally the hardware requires that all linear bitmaps stored in offscreen memory start on an 8 byte or higher boundary, and this value lets the application know how to align the source data.

The *IOPortsTable* field is an offset to a list of all Non-VGA I/O ports that the VBE/AF driver will need access to in order to function correctly. The VBE/AF specifications assumes that the loader/application will provide full I/O access to all the standard VGA I/O ports. The format of the table is:

Port, Port, ... , Port, Terminate Port List with FFFFh.

For example, for the Port/Index combination 1CE/Fh and 13CE/F the table would look like this:

```
01CEh,01CFh,013CEh,013CFh,FFFFh
```

The *IOMemoryBase* field contains the 32 bit physical base addresses pointing to the start of up to 4 separate memory mapped register areas required by the controller. The *IOMemoryLen* field contains the lengths of each of these memory mapped IO areas in bytes. When the application maps the memory mapped IO regions for the driver, the linear address of the mapped memory areas will then be stored in the corresponding entries in the *IOMemMaps* array, and will be used by the driver for accessing the memory mapped registers on the controller. If any of these regions are not required, the *IOMemoryBase* entries will be NULL and do not need to be mapped by the application.

The *BankedMem* field contains the mapped linear address of the banked memory framebuffer, and will be filled in by the application when it has loaded the device driver. This provides the device driver with direct access to the video memory on the controller when in the banked framebuffer modes.

The *LinearMem* field contains the mapped linear address of the linear memory framebuffer, and will be filled in by the application when it has loaded the device driver. This provides the device driver with direct access to the video memory on the controller when in the linear framebuffer modes.

The *Sel0000h* field contains a 32 bit read/write data selector to the entire 1Mb of real mode memory so that the application can directly access important real mode memory locations. This must be provided by the application when it first loads the device driver.

The *Sel0040h* field contains a 32 bit read/write data selector to the 64Kb segment located at segment 40h in real mode memory. This is provided so that code ported from 16 bit directly can directly access locations in the real mode BIOS data area through a standard selector and offset.

The *SelA000h, SelB000h* and *SelC000h* provide 32 bit read/write data selectors to the 64Kb segments located at A000h, B000h and C000h in real mode memory. This gives the application direct access to the standard VGA banked framebuffer memory and the video BIOS ROM segments, so that these locations can be accessed through a selector offset combination.

The device driver state variables maintain information needed for correct operation when multi buffering and for accessing the offscreen memory regions. See Multi Buffering on page 7 and Accessing Offscreen Memory on page 8 for more information.

The remainder of the device driver structure contains the 32 bit near callable device driver functions, and each is documented in detail below.

Device Callback Functions

This section describes in detail each of the device callback functions required by the device driver. These callbacks *must* be provided by the application program prior to calling any of the device driver functions directly. The callback functions will need to be implemented using operating system specific services, and provide the ability for the device driver to communicate with the low level Video BIOS on the video card.

Int86

This device callback function simulates a real mode interrupt, and is usually used for calling the standard VGA Int 10h graphics interrupt handler to initialize video modes etc. This function is defined similarly to the equivalent DPMI function, so in DPMI environments you can simply transfer the call directly to the appropriate DPMI routine. The DPMI register structure is as follows:

```
typedef struct {
    long    edi;
    long    esi;
    long    ebp;
    long    reserved;
    long    ebx;
    long    edx;
    long    ecx;
    long    eax;
    short   flags;
    short   es,ds,fs,gs,ip,cs,sp,ss;
    } AF_DPMI_regs;
```

```
Input:      AX          =   0x300
            BL          =   Interrupt number
            BH          =   0 (reserved)
            CX          =   0 (always)
            DS:EDI      =   Pointer to DPMI register structure

Output:     DS:EDI      =   Pointer to modified register structure
```

CallRealMode

This device callback function simulates a function call to a real mode procedure, and is usually used for making direct calls to the graphics accelerator BIOS. This function is defined the same as the equivalent DPMI function, so in DPMI environments you can simply transfer the call directly to the appropriate DPMI routine. The DPMI register structure is the same as above.

Note that the actual address of the real mode procedure to be called is the stored in the CS:IP fields in the DPMI register structure.

```
Input:      AX          =   0x301
            BH          =   0 (reserved)
            CX          =   0 (always)
            DS:EDI      =   Pointer to DPMI register structure
```

```
Output:     DS:EDI    =  Pointer to modified register structure
```

Configuration Functions

This section describes in detail each of the device driver configuration functions defined by VBE/AF. Access to these function is provided by simply performing a near call to the address of the desired function stored within the device driver function address table. However before any of the configuration functions can be called, the *InitDriver* function must be called immediately after loading the driver to initialize it for correct operation.

InitDriver

This function initializes the VBE/AF device driver context once it has been loaded onto the heap. This initialization function first verifies the presence of the supported display adapter (returning an error if not found). If the appropriate device has been found, it will then build the table of entry points to all the initialization functions in the device driver, so that the application can then call these functions directly. Note that this initialization function does *not* build the table of entry points to the accelerator functions. These entry points will be created when an appropriate video mode is initialized.

See the device context structure in VBEAF.INC on page 49 for more information on the format.

```
Input:      DS:EBX    =  Address of loaded device context

Output:     EAX       =  0  - Initialization successful
                         -1 - Device not detected
```

GetVideoModeInfo

This function returns extended information about a specific VBE/AF display mode from the mode list pointed to by the *AvailableModes* pointer in the device context structure. This function fills the *AF_modeInfo* structure with technical details about the requested mode.

```
Input:      EAX       =  Mode number
            DS:EBX    =  Address of loaded device context
            ES:EDI    =  Address of ModeInfoBlock to fill

Output:     EAX       =  0  - Successful
                         -1 - Invalid video mode
```

The format of the *AF_modeInfo* block is as follows:

```
typedef struct {
    AF_uint16   Attributes;         /* Mode attributes                  */
    AF_uint16   XResolution;        /* Horizontal resolution in pixels  */
    AF_uint16   YResolution;        /* Vertical resolution in pixels    */
    AF_uint16   BytesPerScanLine;   /* Bytes per horizontal scan line   */
    AF_uint16   BitsPerPixel;       /* Bits per pixel                   */
    AF_uint16   MaxBuffers;         /* Maximum num. of display buffers  */

    /* RGB pixel format info */
    AF_uint8    RedMaskSize;        /* Size of direct color red mask    */
```

```
    AF_uint8    RedFieldPosition;   /* Bit posn of lsb of red mask      */
    AF_uint8    GreenMaskSize;      /* Size of direct color green mask  */
    AF_uint8    GreenFieldPosition; /* Bit posn of lsb of green mask    */
    AF_uint8    BlueMaskSize;       /* Size of direct color blue mask   */
    AF_uint8    BlueFieldPosition;  /* Bit posn of lsb of blue mask     */
    AF_uint8    RsvdMaskSize;       /* Size of direct color res mask    */
    AF_uint8    RsvdFieldPosition;  /* Bit posn of lsb of res mask      */

    /* Virtual buffer dimensions */
    AF_uint16   MaxBytesPerScanLine;/* Maximum bytes per scan line      */
    AF_uint16   MaxScanLineWidth;   /* Maximum pixels per scan line     */
    AF_uint8    reserved[118];      /* Pad to 128 byte block size       */
    } AF_modeInfo;
```

The *Attributes* field contains a number of flags that describes certain important characteristics of the graphics mode:

```
#define afHaveMultiBuffer   0x0001
#define afHaveVirtualScroll 0x0002
#define afHaveBankedBuffer  0x0004
#define afHaveLinearBuffer  0x0008
#define afHaveAccel2D       0x0010
#define afHaveDualBuffers   0x0020
#define afHaveHWCursor      0x0040
#define afHave8BitDAC       0x0080
#define afNonVGAMode        0x0100
```

The *afHaveMultiBuffer* flag is used to determine whether the video mode can support hardware multi buffering used for flicker free animation. If this bit is 0, then the application cannot start a multi buffered video mode (usually because there is not enough display memory for two or more video buffers).

The *afHaveVirtualScroll* flag is used to determine if the video mode supports virtual scrolling functions. If this bit is 0, then the application cannot perform virtual scrolling (multi buffering and virtual scrolling are separate, since some controllers may support one but not the other; most support both).

The *afHaveBankedBuffer* flag is used to determine if the video mode supports the banked framebuffer access modes. If this bit is 0, then the application cannot use the banked framebuffer style access. Some controllers may not support a banked framebuffer mode in some modes. In this case a linear framebuffer mode will be provided (either a banked buffer or linear buffer must be available for the mode to be valid).

The *afHaveLinearBuffer* flag is used to determine if the video mode supports the linear framebuffer access modes. If this bit is 0, then the application cannot start the linear framebuffer video mode.

The *afHaveAccel2D* flag is used to determine if the video mode supports 2D accelerator functions. If this bit is 0, then the application can only use direct framebuffer access in this video mode, and the 2D acceleration functions are not available. The cases where this might crop up are more prevalent than you might think. This bit may be 0 for very low resolution video modes on some controllers, and on older controllers for the 24 bit and above video modes.

The *afHaveDualBuffers* flag is used to determine if double buffering is implemented as two distinct framebuffers, or using a single framebuffer and varying the starting display address. If this flag is set, the offscreen memory areas for the video modes will be physical different memory for the two active display buffers, and bitmaps will need to be duplicated in both

buffers. If this flag is not set, the offscreen buffer will be the same physical memory regardless of which buffer is currently active.

The *afHaveHWCursor* flag is used to determine if the controller supports a hardware cursor for the specified video mode. You *must* check this flag for each video mode before attempting to use the hardware cursor functions as some video modes will not be able to support the hardware cursor (but may still support 2D acceleration).

The *afHave8BitDAC* flag is used to determine if the controller will be using the 8 bit wide palette DAC modes when running in 256 color index modes. The 8 bit DAC modes allow the palette to be selected from a range of 16.7 million colors rather than the usual 256k colors available in 6 bit DAC mode. The 8 bit DAC mode allows the 256 color modes to display a full range of 256 grayscales, while the 6 bit mode only allows a selection of 64 grayscales. Note that unlike VBE 2.0 the 8 bit DAC mode is not selectable. If the hardware supports an 8 bit DAC, it will always be used.

The *afNonVGAMode* flag is used to determine if the mode is a VGA compatible mode or a NonVGA mode. If this flag is set, the application software *must* ensure that no attempts are made to directly program *any* of the standard VGA compatible registers such as the RAMDAC control registers and input status registers while the NonVGA graphics mode is used. Attempting to use these registers in NonVGA modes generally results in the application program hanging the system.

The *XResolution* and *YResolution* specify the width and height in pixel elements for this display mode, while the *BytesPerScanLine* field specifies how many full bytes are in each logical scanline. The logical scanline could be equal to or larger than the displayed scanline, and can be changed when the video mode is first initialized.

The *MaxBuffers* field holds the total number of display buffers available in that graphics mode. When initializing multi buffered video modes, the number of buffers requested cannot exceed this value.

The *RedMaskSize*, *GreenMaskSize*, *BlueMaskSize*, and *RsvdMaskSize* fields define the size, in bits, of the red, green, and blue components of an RGB pixel. A bit mask can be constructed from the MaskSize fields using simple shift arithmetic. For example, the MaskSize values for an RGB 5:6:5 mode would be 5, 6, 5, and 0, for the red, green, blue, and reserved fields, respectively.

The *RedFieldPosition*, *GreenFieldPosition*, *BlueFieldPosition*, and *RsvdFieldPosition* fields define the bit position within the RGB 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 an RGB 5:6:5 mode would be 11, 5, 0, and 0, for the red, green, blue, and reserved fields, respectively.

The *MaxBytesPerScanLine* and *MaxScanLineWidth* fields define the maximum virtual framebuffer coordinates that can be initialized for the mode, in both bytes and pixels. If an attempt is made to initialize a graphics mode with values larger than these values, the mode set will fail.

SetVideoMode

This function is used to initialize a specified video mode. The mode number passed to this function should be one stored in the *AvailableModes* tables stored in the device context buffer, or the mode set will fail.

```
Input:      EAX       =  Mode number
            DS:EBX    =  Address of loaded device driver
            ECX       =  Scanline width in bytes (-1 for default)
            EDX       =  Virtual X resolution (if afVirtualScroll set)
            ESI       =  Virtual Y resolution (if afVirtualScroll set)
            EDI       =  Total num. of display buffers (if afMultiBuffer set)

Output:     EAX       =  0  - Mode set successfully
                             -1 - Mode set failed
            ECX       =  Actual scanline width for mode
```

When the video mode is initialized, you can pass in a specific scanline width that should be used in the ECX register. If you pass in a value of -1, the default scanline width for that video mode will be used, and this value will be returned in the ECX register. If you pass in a value other than -1, the driver will attempt to satisfy your request with the next largest value that the controller can actually handle, and the actual value programmed will be returned in the ECX register. It is possible that the video mode cannot have the scanline width changed, or that the scanline width requested was too large for the video mode. In this case the mode set function will fail. This is most useful for initializing the 24 bit video modes with even 1Kb multiples that divide 64Kb evenly when in the banked framebuffer modes, to ensure that bank boundaries do not occur in the middle of a normal pixel.

After the video mode has been initialized, any left over video memory will be enabled for use as offscreen video memory. If there is not enough offscreen memory available, the *OffscreenOffset* variable of the device context block will be set to 0. Note that not all left over video memory can be used by the application as offscreen memory, as the driver may need to maintain a number of small buffers at the end of video memory for storing the hardware cursor definition and pattern fill images.

This function also accepts the following flags logically OR'ed in with the passed video mode number, to change the way that the selected video mode is initialized:

```
#define afDontClear     0x8000
#define afLinearBuffer  0x4000
#define afMultiBuffer   0x2000
#define afVirtualScroll 0x1000
```

The *afDontClear* flag is used to specify that the video memory should not be cleared when the video mode is initialized. By default the video memory will be cleared to all 0's by the device driver.

The *afLinearBuffer* flag is used to specify that the application wishes to enable the linear framebuffer version of the video mode. On many controllers, the banked and linear framebuffers cannot be accessed at the same time. Also note that on many new PCI controllers, PCI burst mode is only enabled in the linear framebuffer modes, so these modes should be used whenever possible for maximum performance. Make sure that you check the *afHaveLinearBuffer* flag in the mode attributes field to determine if this is supported in the selected video mode.

The *afMultiBuffer* flag must be set if the application intends to use multi buffering in the video mode, and the total number of display buffers to be allocated is passed in the EDI register. When the video mode is initialized however, the active and visible buffers will both be set to 0. Multi buffering can be enabled by setting the active and visual buffers to different values. Make sure that you check the *afHaveMultiBuffer* flag in the mode attributes field to determine if the selected video mode supports multi buffering. If there is not enough video memory available for all display buffers or the number of buffers requested is larger than the maximum available for the requested mode, the mode set will fail.

The *afVirtualScroll* flag can be set to enable a virtual scrolling mode. The application will also need to pass in a valid virtual screen resolution to be used for the video mode. If there is not enough memory for the virtual mode, the mode set function will fail. The function will also fail if the video mode does not support virtual scrolling. For virtual scrolling modes, if you pass a value of -1 for the scanline width to be used, the next largest scanline width that supports the desired X resolution for the virtual video mode will be programmed and this value will be returned in ECX. You can override the scanline width if you desire to obtain a value other than the default (such as even 1Kb scanline widths in 24 bit modes).

RestoreTextMode

This function restores the hardware device to the standard 80x25 text mode. This function must be called after using an accelerated video mode to return the system back to the default state. It is not sufficient to call the standard BIOS text mode function, as some accelerator systems have different hardware devices for the standard text mode and accelerated graphics mode operations. It can be assumed that after this function has been executed, the system will be back in the standard VGA text mode.

```
Input:      DS:EBX    =  Address of loaded device context
```

Direct Framebuffer Access Functions

This section describes the set of VBE/AF device driver functions related to directly accessing the framebuffer memory, through either a banked or linear framebuffer.

SetBank32

This is a special fully relocateable, 32 bit function that changes the currently active read/write bank for banked framebuffer modes, and can be copied and called from any function. Only a single read/write bank is supported by VBE/AF devices, and it may be either 4Kb or 64Kb in length. The primary purpose of this function is to provide a 32 bit relocateable routine that can be used to virtualize the framebuffer under Windows using the DVA.386 or VFLATD.386 virtual framebuffer device drivers. The *SetBank32Len* variable contains the length of this function, including the near return statement at the end. Note that this function can *only* be supported if the controller uses memory mapped registers for programming the bank value, and for devices that use memory mapped IO exclusively, this function will be a NULL. However these devices all support a real hardware linear framebuffer mode anyway, so the buffer will never need to be virtualized in software.

```
Input:      EDX       =  New bank number
```

SetBank

This function changes the currently active read/write bank for banked framebuffer modes. This allows an application to directly access the video framebuffer through a small memory aperture window. Only a single read/write bank is supported by VBE/AF devices, and it may be either 4Kb or 64Kb in length.

```
Input:      EDX      =  New bank number
            DS:EBX   =  Address of loaded device driver
```

WaitTillIdle

This function pauses execution until the hardware accelerator has completed all currently queued rendering operations. The primary purpose of this function is to provide the application with the ability to ensure all rendering is complete, before swapping display pages when doing double buffering, or before directly accessing the framebuffer memory.

```
Input:      DS:EBX   =  Address of loaded device driver
```

EnableDirectAccess

This function disables the accelerator and turns on direct framebuffer access. The primary purpose of this function is to correctly arbitrate video memory access between the accelerator and the application. You *must* call this function before you perform any direct rendering to the video memory if the entry in the device context is not NULL. If the entry is NULL, then the controller does not need to arbitrate access and this function should not be called.

```
Input:      DS:EBX   =  Address of loaded device driver
```

DisableDirectAccess

This function disables direct framebuffer access and turns on the hardware accelerator again. The primary purpose of this function is to correctly arbitrate video memory access between the accelerator and the application. You *must* call this function before you perform any accelerated rendering again if the *EnableDirectAccess* function was not NULL.

```
Input:      DS:EBX   =  Address of loaded device driver
```

Virtual Scrolling Functions

SetDisplayStart

This function sets the start of the currently visible area of video memory to a specified starting pixel coordinate in video memory. This is useful for games that provide virtual scrolling or for large virtual desktops for GUI applications. If both virtual scrolling and double buffering are enabled at the same time, the display starting address is the logical pixel coordinate within the visible framebuffer. If you wish to change the display start and then change the visible buffer, you should call this function with EAX set to -1 then call the *SetVisibleBuffer* function as per normal.

```
Input:      EAX      =  1  - Wait for vertical retrace
```

```
                    0  - Don't wait for vertical retrace
                    -1 - Just set the coordinates, don't update CRT start
            DS:EBX  =  Device Context Buffer
            ECX     =  Display start X coordinate
            EDX     =  Display start Y coordinate
```

Multi Buffering Functions

SetActiveBuffer

This function sets the currently active output buffer in multi buffered modes. The active buffer and visible may be the same, which essentially is the same as running in single buffer mode. Changing the active buffer automatically updates the *OriginOffset, OffscreenStartY* and *OffscreenEndY* device driver state variables. See Multi Buffering on page 7 for more information.

```
Input:      EAX     =  Buffer index (0 to MaxBuffers)
            DS:EBX  =  Device Context Buffer
```

SetVisibleBuffer

This function sets the currently visible buffer in multi buffered modes. The active and visible buffers may be the same, which is essentially the same as running in single buffer mode. The vertical retrace flag can be used to enable or disable waiting for a vertical retrace when the visible buffer is swapped. Generally you need to wait for a vertical retrace to obtain flicker free animation in double buffered modes, but sometimes it is useful to disable it to be able to measure the real throughput of certain rendering commands during optimization work.

```
Input:      EAX     =  Buffer index (0 to MaxBuffers)
            DS:EBX  =  Device Context Buffer
            EDX     =  1 - Wait for vertical retrace
                       0 - Don't wait for vertical retrace
```

Palette Functions

SetPaletteData

This function programs the color palette information for the current video mode, and is only valid in 8 bit color index modes. Color palette information is passed to the function in the an array of *AF_palette* structures, which is in a format similar to the Windows RGBQUAD structure, with 8 bits per color channel. Note that this is different to the standard VGA palette programming routines, which normally take values with 6 bits per color channel. Internally the VBE/AF driver will convert the 8 bit palette values to 6 bits per primary if this is what the underlying hardware supports. The *AF_palette* structure is defined as follows:

```
typedef struct {
    AF_uint8    blue;               /* Blue component of color          */
    AF_uint8    green;              /* Green component of color         */
    AF_uint8    red;                /* Blue component of color          */
    AF_uint8    alpha;              /* Alpha or alignment byte          */
    } AF_palette;
```

The wait for vertical retrace flag is used to synchronize the palette update with the start of the vertical retrace. However if you are changing palette values at the same time as swapping display pages, you may want to disable vertical retrace synching and program the palette entries directly *after* swapping display pages. Generally you need to synchronize with the vertical retrace while programming the palette to avoid the onset of snow.

```
Input:      EAX       =  1 - Wait for vertical retrace
                         0 - Don't wait for vertical retrace
            DS:EBX    =  Device Context Buffer
            ECX       =  Number of entries to program
            EDX       =  Starting index to program
            DS:ESI    =  Pointer to palette data to program
```

SetGammaCorrectionData

This function programs the gamma correction tables for 15 bit and above video modes. The gamma correction tables are used in these video modes to adjust the response curves of each of the three color guns for color matching purposes. The gamma correction tables are assumed to be 256 entries deep with three independent channels for each of red, green and blue, with 8 bits of intensity for each color channel ramp. Gamma correction data is passed to the function in an array of *AF_palette* structures, similar to the palette setting routine above. See SetPaletteData above for the format of the *AF_palette* structure.

```
Input:      DS:EBX    =  Device Context Buffer
            ECX       =  Number of entries to program
            EDX       =  Starting index to program
            DS:ESI    =  Pointer to color correction data to program
```

Hardware Cursor Functions

SetCursor

This function downloads the specified cursor definition from the application into the hardware cursor. The cursor data is passed by the application in the following *AF_cursor* format, which is similar to the Windows 3.1 cursor file format:

```
typedef struct {
    AF_uint32   xorMask[32];        /* Cursor XOR mask                */
    AF_uint32   andMask[32];        /* Cursor AND mask                */
    AF_uint32   hotx;               /* Cursor X coordinate hot spot   */
    AF_uint32   hoty;               /* Cursor Y coordinate hot spot   */
    } AF_cursor;
```

```
Input:      DS:EBX    =  Device Context Buffer
            DS:ESI    =  Pointer to 32x32 Cursor Data
```

SetCursorPos

This function sets the location of the hardware cursor in display coordinates. This function takes the X and Y coordinates of the new cursor location. This function will place the cursor so that the hotspot of the cursor image is located at the specified (X,Y) location, and will correctly handle special cases where the cursor definition needs to be located off the edges of the display screen (such as when X=0, Y=0 and HotX,HotY > 0).

```
Input:      EAX       =  X coordinate of cursor
```

```
            DS:EBX    =  Device Context Buffer
            ECX       =  Y coordinate of cursor
```

SetCursorColor

This function sets the current hardware cursor color. In 8 bit color index modes, the color index to use for the cursor color is passed in the AL register. However in all other DirectColor modes (15/16/24 and 32 bits per pixel modes) the color values are passed in as individual 8 bit Red, Green and Blue components.

```
Input:      AL        =  8 bit index, or RGB red component
            AH        =  RGB green component
            DS:EBX    =  Device Context Buffer
            CL        =  RGB blue component
```

ShowCursor

This function unconditionally either show or hides the hardware cursor.

```
Input:      EAX       =  1  - Show cursor
                         0  - Hide cursor
            DS:EBX    =  Device Context Buffer
```

2D Accelerator Drawing Context Functions

SetMix

This function sets the current foreground and background mix operation for subsequent accelerated rendering primitives. The mix does not change that often, so it is only set once for a range of rendering primitives. The following mix modes are currently supported by VBE/AF:

```
00h     AF_REPLACE_MIX Source replaces dest
01h     AF_XOR_MIX            Source XOR dest
02h     AF_OR_MIX             Source OR dest
03h     AF_AND_MIX            Source AND dest
04h     AF_NOP_MIX            Destination unchanged
```

The foreground mix is normally the mix that is used for all the solid filling functions. However for patterned fill functions, the foreground mix is used for all pixels where the pattern is a 1, and the background mix is used for all pixels where the pattern is a 0. Setting the foreground or background mix to AF_NOP_MIX allows you to render primitives with a transparent pattern.

```
Input:      EAX       =  New foreground mix
            DS:EBX    =  Device Context Buffer
            ECX       =  New background mix
```

Set8x8MonoPattern

This function sets up an 8x8 monochrome pattern for all subsequent pattern filled functions. This function downloads the specified 8x8 bitmap fill pattern for rectangle and scanline filling, where the pattern is X and Y coordinate aligned with the left edge and top edge of the display. Thus the bit in the stipple pattern that applies to a specific pixel in the scanline is determine by AND'ing the pixel's X coordinate with 8. Hence pixel 0 corresponds to bit 0 in byte 0, pixel 1 = bit 1 in byte 0, ... pixel 8 = bit 0 in byte 1 etc. Where a bit is 1 in the bitmap pattern, the pixel is drawn and where it is 0, the pixel remains untouched. It is the responsibility of the calling application to rotate the pattern before calling this routine if it is desired that the pattern be aligned to a

different starting coordinate (such as with Windows Bitmaps and setting the bitmap origin). The bitmap pattern is passed as a packed array of 8 bytes.

```
Input:      DS:EBX    =  Device Context Buffer
            DS:ESI    =  Pointer to 8 bytes for pattern
```

Set8x8ColorPattern

This function sets up an 8x8 color pattern for all subsequent color pattern filled functions. This function downloads the specified 8x8 color fill pattern for rectangle and scanline filling, where the pattern is X and Y coordinate aligned with the left edge and top edge of the display. Thus the colors in the pattern that applies to a specific pixel in the scanline is determine by the pixel's X starting at the left. Hence pixel 0 corresponds to color 0, pixel 1 = color 1 etc. It is the responsibility of the calling application to rotate the pattern before calling this routine if it is desired that the pattern be aligned to a different starting coordinate (such as with Windows Bitmaps and setting the bitmap origin). The color pattern is passed as an 8x8 array of DWORDS, one for each pixel in the pattern. Each pixel color value is packed for the appropriate display mode and will always take up an entire DWORD (i.e.: in 8 bit modes only the bottom 8 bits are used). The VBE/AF driver will convert the pattern to the format that it requires internally when it is downloaded to the hardware.

```
Input:      DS:EBX    =  Device Context Buffer
            DS:ESI    =  Pointer to 8x8 color pattern (64 DWORDS)
```

SetLineStipple

This function sets up a 16 bit line stipple for all subsequent stippled line drawing functions. Where a bit is 1 in the stipple pattern, the pixel is drawn and where it is 0, the pixel remains untouched. It is the responsibility of the calling application to rotate the pattern before calling this routine if it is desired that the pattern be aligned to a different starting coordinate.

```
Input:      AX        =  16 bit stipple pattern
            DS:EBX    =  Device Context Buffer
```

SetClipRect

This function sets the current hardware clipping rectangle for subsequent accelerated rendering primitives. All subsequent accelerated output will then be clipped to this clipping rectangle. The parameters for the clip rectangle to be set are passed in the following structure:

```
typedef struct {
    AF_int32   minX;
    AF_int32   minY;
    AF_int32   maxX;
    AF_int32   maxY;
    } AF_clipRect;
```

Note that the maximum coordinates are inclusive, so that for an unclipped 640x480 framebuffer you would set *maxX* to 639 and *maxY* to 479.

```
Input:      DS:EBX    =  Device Context Buffer
            DS:ESI    =  Pointer to AF_clipRect structure
```

2D Accelerator Drawing Functions

DrawScan

This function renders a solid scanline at the specified location in the specified color and currently active foreground mix. This routine will render a scanline from X1 to X2 (exclusive) at the specified Y coordinate. For scan lines where X2 < X1, the X1 and X2 coordinates will be swapped, and for scan lines where X1 = X2, the scanline will be skipped and nothing will be drawn. Note that the pixel at the X2 coordinate passed will *not* be drawn. This function will always be provided by accelerated VBE/AF drivers, and will be implemented with whatever hardware rendering function provides the fastest possible method of rendering scan lines with the installed hardware.

```
Input:      EAX       =  Y coordinate of scan
            DS:EBX    =  Device Context Buffer
            ECX       =  X1 coordinate of scan
            EDX       =  X2 coordinate of scan
            ESI       =  Color to draw in
```

DrawPattScan

This function renders a patterned scanline at the specified location in the specified color and currently active foreground and background mixes. This routine will render a scanline from X1 to X2 (exclusive) at the specified Y coordinate. For scan lines where X2 < X1, the X1 and X2 coordinates will be swapped, and for scan lines where X1 = X2, the scanline will be skipped and nothing will be drawn. Note that the pixel at the X2 coordinate passed will *not* be drawn. The scanline is filled in with the currently active pattern set by calling the *Set8x8MonoPattern* routine.

```
Input:      EAX       =  Y coordinate of scan
            DS:EBX    =  Device Context Buffer
            ECX       =  X1 coordinate of scan
            EDX       =  X2 coordinate of scan
            ESI       =  Foreground color to draw in
            EDI       =  Background color to draw in
```

DrawColorPattScan

This function renders a color patterned scanline at the specified location in the currently active foreground mix. This routine will render a scanline from X1 to X2 (exclusive) at the specified Y coordinate. For scan lines where X2 < X1, the X1 and X2 coordinates will be swapped, and for scan lines where X1 = X2, the scanline will be skipped and nothing will be drawn. Note that the pixel at the X2 coordinate passed will *not* be drawn. The scanline is filled in with the currently active color pattern set by calling the *Set8x8ColorPattern* routine.

```
Input:      EAX       =  Y coordinate of scan
            DS:EBX    =  Device Context Buffer
            ECX       =  X1 coordinate of scan
            EDX       =  X2 coordinate of scan
```

DrawScanList

This function renders a list of solid scan lines starting at the specified location in the specified color and currently active foreground mix. The scanline coordinates are passed as an array of 16

bit integer coordinates, packed with the X1 coordinate followed by the X2 coordinate and so on. For each scanline in the list, this routine will render a scanline from X1 to X2 (exclusive) at increasing Y coordinates. For scan lines where X2 < X1, the X1 and X2 coordinates will be swapped, and for scan lines where X1 = X2, the scanline will be skipped and nothing will be drawn. This function will always be provided by accelerated VBE/AF drivers, and will be implemented with whatever hardware rendering function provides the fastest possible method of rendering scan lines with the installed hardware. It is also one of the workhorse functions that will be used by high level rendering code for drawing non-polygonal solid shapes (ellipses, wedges, regions etc.).

```
Input:      EAX       =  Y coordinate of first scanline
            DS:EBX    =  Device Context Buffer
            ECX       =  Length of scanline list
            EDX       =  Color to draw in
            DS:ESI    =  Pointer to list of 16 bit integer scanline
coordinates
```

DrawRect

This function renders a solid rectangle at the specified location and color in the currently active foreground mix. This routine will render a rectangle from (Left, Top) to (Left+Width-1, Height+Bottom-1) inclusive. This function will always be provided by accelerated VBE/AF drivers, and will be implemented with whatever hardware rendering function provides the fastest possible method of rendering rectangles with the installed hardware. The parameters for the rectangle to be rendered are passed in the following structure:

```
typedef struct {
    AF_int32      left;
    AF_int32      top;
    AF_int32      width;
    AF_int32      height;
    } AF_rect;
```

```
Input:      EAX       =  Color to draw in
            DS:EBX    =  Device Context Buffer
            DS:ESI    =  Pointer to AF_rect structure
```

DrawPattRect

This function renders a patterned rectangle at the specified location and color in the currently active foreground and background mixes. This routine will render a rectangle from (Left, Top) to (Left+Width-1, Height+Bottom-1) inclusive. The rectangle is filled in with the currently active pattern set by calling the *Set8x8MonoPattern* routine. If the hardware supports pattern fills (even if just scanline fills) this function will always be provided by VBE/AF and will provide the fastest way to render pattern filled rectangles. The parameters for the rectangle to be rendered are passed in the following structure:

```
typedef struct {
    AF_int32      left;
    AF_int32      top;
    AF_int32      width;
    AF_int32      height;
    } AF_rect;
```

```
Input:      EAX       =  Foreground color to draw in
            DS:EBX    =  Device Context Buffer
            ECX       =  Background color to draw in
            DS:ESI    =  Pointer to AF_rect structure
```

DrawColorPattRect

This function renders a color patterned rectangle at the specified location in the currently active foreground mix. This routine will render a rectangle from (Left, Top) to (Left+Width-1, Height+Bottom-1) inclusive. The rectangle is filled in with the currently active color pattern set by calling the *Set8x8ColorPattern* routine. If the hardware supports pattern fills (even if just scanline fills) this function will always be provided by VBE/AF and will provide the fastest way to render color pattern filled rectangles. The parameters for the rectangle to be rendered are passed in the following structure:

```
typedef struct {
    AF_int32      left;
    AF_int32      top;
    AF_int32      width;
    AF_int32      height;
    } AF_rect;

Input:       DS:EBX    =  Device Context Buffer
             DS:ESI    =  Pointer to AF_rect structure
```

DrawLine

This function renders a solid line at the specified location and color in the currently active foreground mix. This routine will render a line from (x1,y1) to (x2,y2) inclusive. Note that the coordinates passed to this routine are in 16.16 fixed point format. The parameters for the rectangle to be rendered are passed in the following structure:

```
typedef struct {
    AF_fix32      x1;
    AF_fix32      y1;
    AF_fix32      x2;
    AF_fix32      y2;
    } AF_line;

Input:       EAX       =  Color to draw in
             DS:EBX    =  Device Context Buffer
             DS:ESI    =  Pointer to AF_line structure
```

DrawStippleLine

This function renders a stippled line at the specified location and color in the currently active foreground mix. This routine will render a line from (x1,y1) to (x2,y2) inclusive. Note that the coordinates passed to this routine are in 16.16 fixed point format. The parameters for the rectangle to be rendered are passed in the following structure:

```
typedef struct {
    AF_fix32      x1;
    AF_fix32      y1;
    AF_fix32      x2;
    AF_fix32      y2;
    } AF_line;

Input:       EAX       =  Foreground color to draw in
             DS:EBX    =  Device Context Buffer
             ECX       =  Background color to draw in
             DS:ESI    =  Pointer to AF_line structure
```

DrawTrap

This function renders a solid, flat topped and bottomed trapezoid in the specified color. The parameters for the trapezoid to be rendered are passed in the following structure (note that all coordinates are in 16.16 fixed point format):

```
typedef struct {
    AF_int32      y;
    AF_int32      count;
    AF_fix32      x1;
    AF_fix32      x2;
    AF_fix32      slope1;
    AF_fix32      slope2;
    } AF_trap;
```

Note that this function will always be provided, and will be the fallback polygon rendering function for devices that do not have hardware triangle or quad filling, and will draw the trapezoid by rendering each of the individual scan lines. After this function has been called, the VBE/AF driver will have updated the *y*, *x1* and *x2* variables in the *AF_trap* structure to reflect the final values after scan converting the trapezoid. This ensures that the high level code can properly join up connected trapezoids to complete the rendering of a larger more complex polygon. The standard algorithm for implementing this is C is as follows (note that it handles edges that can cross within the trapezoid properly):

```
while (count--) {
    ix1 = FIXROUND(x1);
    ix2 = FIXROUND(x2);
    if (ix2 < ix1)
        SWAP(ix1,ix2);
    if (ix1 < ix2)
        scanLine(y,ix1,ix2);
    x1 += slope1;
    x2 += slope2;
    y++;
    }
```

```
Input:      EAX       =  Color to draw in
            DS:EBX    =  Device Context Buffer
            DS:ESI    =  Pointer to AF_trap structure
```

DrawTri

This function renders a 2D flat shaded triangle in the specified color given three fixed point coordinates. The parameters for the triangle to be rendered are passed in the following structure:

```
typedef struct {
    AF_fxpoint    *v1;
    AF_fxpoint    *v2;
    AF_fxpoint    *v3;
    AF_fix32      xOffset;
    AF_fix32      yOffset;
    } AF_triangle;
```

The structure contains pointers to the three vertices in the triangle in the *AF_fxpoint* structure, which represent each vertex in 16.16 fixed point format. The format of the *AF_fxpoint* structure is as follows:

```
typedef struct {
    AF_fix32      x;
    AF_fix32      y;
    } AF_fxpoint;
```

The x and y coordinate offsets are added to every vertex in the triangle before being sent to the hardware for performing viewport transformations. Refer to the section above on the 2D coordinate system for more information.

```
Input:      EAX       =  Color to draw in
            DS:EBX    =  Device Context Buffer
            DS:ESI    =  Pointer to AF_triangle structure
```

DrawQuad

This function renders a 2D flat shaded quadrilateral in the specified color given four fixed point coordinates. The parameters for the quadrilateral to be rendered are passed in the following structure:

```
typedef struct {
    AF_fxpoint    *v1;
    AF_fxpoint    *v2;
    AF_fxpoint    *v3;
    AF_fxpoint    *v4;
    AF_fix32      xOffset;
    AF_fix32      yOffset;
    } AF_quad;
```

The structure contains pointers to the four vertices in the triangle in the *AF_fxpoint* structure, which represent each vertex in 16.16 fixed point format. The format of the *AF_fxpoint* structure is as follows:

```
typedef struct {
    AF_fix32      x;
    AF_fix32      y;
    } AF_fxpoint;
```

The x and y coordinate offsets are added to every vertex in the triangle before being sent to the hardware for performing viewport transformations. Refer to the section above on the 2D coordinate system for more information.

```
Input:      EAX       =  Color to draw in
            DS:EBX    =  Device Context Buffer
            DS:ESI    =  Pointer to AF_quad structure
```

PutMonoImage

This function copies a monochrome bitmap image from a system memory buffer to video memory using the hardware accelerator, which is used for fast bitmap masking and font rendering operations. The bitmap is rendered in the specified color using the currently active foreground and background mixes. The parameters for the image to be transferred are passed in the following structure:

```
typedef struct {
    AF_int32      x;
    AF_int32      y;
    AF_int32      byteWidth;
    AF_int32      height;
    void          *image;
    } AF_putMonoImage;
```

The *x* and *y* parameters define the destination coordinate for the image, and the *byteWidth* and *height* parameters define the dimensions of the monochrome image to be transferred. The *image* pointer points to the start of the monochrome image in system memory and is byte packed and need not contain any padding for each scanline (Windows monochrome BMP's are padded to a DWORD boundary for each scanline).

```
Input:      EAX       =  Foreground color to draw in
            DS:EBX    =  Device Context Buffer
            ECX       =  Background color to draw in
            DS:ESI    =  Pointer to AF_putMonoImage structure
```

BitBlt

This function copies a rectangular region of video memory from one location to another. The parameters for the block to be copied are passed in the following structure:

```
typedef struct {
    AF_int32      left;
    AF_int32      top;
    AF_int32      width;
    AF_int32      height;
    AF_int32      dstLeft;
    AF_int32      dstTop;
    AF_int32      op;
    } AF_bitBlt;
```

This routine will copy a rectangular region of video memory from (left, top, left+width-1, top+height-1) to (dstLeft, dstTop) within video memory with the specified mix mode, and will also correctly handle cases of overlapping regions in video memory.

```
Input:      DS:EBX    =  Device Context Buffer
            DS:ESI    =  Pointer to AF_bitBlt structure
```

BitBltLin

This function copies a linear region of video memory from the offscreen buffer to a rectangular region in the active buffer. This version is different to the above version in that the source region to be copied is non-conforming, and can have a different logical scanline width to the destination region. This allows the bitmaps to be stored contiguously in offscreen memory, rather than requiring the offscreen memory to be divided up into rectangular regions. The parameters for the block to be copied are passed in the following structure:

```
typedef struct {
    AF_int32      srcOfs;
    AF_int32      dstLeft;
    AF_int32      dstTop;
    AF_int32      width;
    AF_int32      height;
    AF_int32      op;
    } AF_bitBltLin;
```

This routine will copy a linear region of video memory from *srcOfs* from the start of video memory with a byte width of *srcByteWidth* bytes to the destination rectangle (dstLeft, dstTop, dstLeft+width-1, dstTop+height-1) with the specified mix mode. Note that the value of *srcOfs* must be aligned to the boundary specified in the *LinearGranularity* field of the device context buffer. The results of this routine are undefined if the video memory regions overlap.

```
Input:      DS:EBX    =  Device Context Buffer
            DS:ESI    =  Pointer to AF_bitBltLin structure
```

TransBlt

This function copies a rectangular region of video memory from one location to another with source transparency. The parameters for the block to be copied are passed in the following structure:

```
typedef struct {
    AF_int32      left;
    AF_int32      top;
    AF_int32      width;
    AF_int32      height;
    AF_int32      dstLeft;
    AF_int32      dstTop;
```

```
    AF_int32      op;
    AF_color      transparent;
    } AF_transBlt;
```

This routine will copy a rectangular region of video memory from (left, top, left+width-1, top+height-1) to (dstLeft, dstTop) within video memory with the specified mix mode and with source transparency. The transparent color passed will be used to mask out pixels in the source image from being written to the destination area. This results of this routine are undefined if the source and destination rectangles overlap.

Input: DS:EBX = Device Context Buffer
DS:ESI = Pointer to *AF_transBlt* structure

TransBltLin

This function copies a linear region of video memory from the offscreen buffer to a rectangular region in the active buffer with source transparency. This version is different to the above version in that the source region to be copied is non-conforming, and can have a different logical scanline width to the destination region. This allows the bitmaps to be stored contiguously in offscreen memory, rather than requiring the offscreen memory to be divided up into rectangular regions. The parameters for the block to be copied are passed in the following structure:

```
typedef struct {
    AF_int32      srcOfs;
    AF_int32      dstLeft;
    AF_int32      dstTop;
    AF_int32      width;
    AF_int32      height;
    AF_int32      op;
    AF_color      transparent;
    } AF_transBltLin;
```

This routine will copy a linear region of video memory from *srcOfs* from the start of video memory with a byte width of *srcByteWidth* bytes to the destination rectangle (dstLeft, dstTop, dstLeft+width-1, dstTop+height-1) with the specified mix mode and with source transparency. Note that the value of *srcOfs* must be aligned to the boundary specified in the *LinearGranularity* field of the device context buffer. The transparent color passed will be used to mask out pixels in the source image from being written to the destination area. The results of this routine are undefined if the video memory regions overlap.

Input: DS:EBX = Device Context Buffer
DS:ESI = Pointer to *AF_transBltLin* structure

Questions & Answers

Is this specification only useful for DOS applications?

No. This specification defines how to access acceleration features at the very lowest level of abstraction. As such, any software that runs on the x86 platform can conceivably use this information. This includes DOS applications, games and "plug and play" operating system drivers for environments such as Windows, OS/2, UNIX and others. This specification can be used by any application or operating system that needs access to high performance accelerator features. It will be especially useful for entertainment, virtual reality and other applications that exert high demands on the graphics subsystem and thus need direct access to accelerator functions.

Is this just another software layer to slow things down?

No. VBE/AF was designed to do three things: 1.) Let the application software or operating system know what features are available *in hardware*. 2.) Provide the code to enable those features and 3.) Get out of the way and let the software write directly to the hardware. This is in contrast with a higher level API that does many complex functions in software. With VBE/AF, a programmer could choose to use one of those higher level APIs that would in turn call VBE/AF for device support, or they can write directly to the VBE/AF specification using their own proprietary routines and libraries.

Why have you implemented only a small set of functions?

The VBE/AF functions have been boiled down into the smallest possible set of accelerator functions that can be implemented as efficiently as possible and are the most commonly needed by application software. When an application calls the VBE/AF device driver routines, it will already have performed any kind of application specific special casing necessary.

Will any additional hardware be necessary to support VBE/AF?

No. Actually VBE/AF can be implemented on most graphics cards shipped in the last 3-4 years. If a particular VBE/AF function is not supported in hardware, the VBE/AF implementation will just return back that that function is not supported; all other functions can be accessed normally.

Appendix A - Sample C API

This appendix contains sample source code for a simple C based API for loading the VBE/AF driver file and calling the accelerated rendering functions from C. Note that for efficiency, a high performance graphics library would call performance sensitive primitive rendering functions directly from assembler for speed, rather than using the C based functions.

Some of the sample code provided in this Appendix depends on some freely available software tools developed by SciTech Software, such as the PM/Pro library for interfacing with DOS extender and operating system specific functions. All functions preceded with *PM_* are part of this package. For more information and the source code to the PM/Pro functions, you can download the library from SciTech Software's ftp site, ftp.scitechsoft.com under the devel directory.

VBEAF.H

```
/****************************************************************************
*
*                VESA BIOS Extensions/Accelerator Functions
*                            Version 1.0
*
*                   Copyright (C) 1996 SciTech Software.
*                           All rights reserved.
*
* Filename:     $Workfile:   vbeaf.h  $
* Developed by: SciTech Software
*
* Language:     ANSI C
* Environment:  IBM PC 32 bit Protected Mode.
*
* Description:  Header file for the VBE/AF Graphics Acclerator Driver API.
*
*               When this code is used to load the VBEAF.DRV driver file,
*               it will look for it in the following standard locations
*               in the following order:
*
*                   1. C:\VBEAF.DRV for DOS, Windows, OS/2
*                        /VBEAF.DRV for Unix
*                   2. VBEAF_PATH environment variable
*                   3. Path passed to AF_loadDriver
*
*               The last location searched is to allow specific versions
*               of a driver file to be stored with a applications in case
*               an application needs a specific version for some obscure
*               reason.
*
* $Date:   21 Feb 1996 18:35:46  $ $Author:   KendallB  $
*
****************************************************************************/

#ifndef __VBEAF_H
#define __VBEAF_H

#ifndef __DEBUG_H
#include "debug.h"
#endif

/*---------------------- Macros and type definitions ----------------------*/

/* Define the calling conventions for the code in this module */
```

```
#define AFAPI  _ASMAPI          /* 'C' calling conventions always       */

#pragma pack(1)

/* Type definitions for fundamental types */

typedef char            AF_int8;    /* 8 bit signed value               */
typedef short           AF_int16;   /* 16 bit signed value              */
typedef long            AF_int32;   /* 32 bit signed value              */
typedef unsigned char   AF_uint8;   /* 8 bit unsigned value             */
typedef unsigned short  AF_uint16;  /* 16 bit unsigned value            */
typedef unsigned long   AF_uint32;  /* 32 bit unsigned value            */
typedef long            AF_fix32;   /* 16.16 signed fixed point format  */

typedef AF_uint8    AF_pattern;     /* Pattern array elements           */
typedef AF_uint16   AF_stipple;     /* 16 bit line stipple pattern      */
typedef AF_uint32   AF_color;       /* Packed color values              */

/* VBE/AF Graphics Accelerator Driver structure.
 *
 * Internally in the structure there are members represented as pointers.
 * However when the driver file is first loaded, these values will actually
 * be offsets from the start of the loaded driver file, but the intial
 * call to InitDriver will 'fix-up' the pointers and turn them into
 * real pointers.
 */

typedef struct {
    /*-------------------------------------------------------------------*/
    /* Device driver header block                                        */
    /*-------------------------------------------------------------------*/

    char        Signature[12];      /* 'VBEAF.DRV\0' 12 byte signature  */
    AF_uint32   Version;            /* Driver Interface Version (1.0)   */
    AF_uint32   DriverRev;          /* Driver revision number           */
    char        OemVendorName[80];  /* Vendor Name string               */
    char        OemCopyright[80];   /* Vendor Copyright string          */
    AF_int16    *AvailableModes;    /* Offset to supported mode table   */
    AF_uint32   TotalMemory;        /* Amount of memory in Kb detected  */
    AF_uint32   Attributes;         /* Driver attributes                */
    AF_uint32   BankSize;           /* Bank size in Kb (4Kb or 64Kb)    */
    AF_uint32   BankedBasePtr;      /* Physical addr of banked buffer   */
    AF_uint32   LinearSize;         /* Linear buffer size in Kb         */
    AF_uint32   LinearBasePtr;      /* Physical addr of linear buffer   */
    AF_uint32   LinearGranularity;  /* Linear blt granularity in bytes  */
    AF_uint16   *IOPortsTable;      /* Offset of I/O ports table        */
    AF_uint32   IOMemoryBase[4];    /* Base address of I/O memory maps  */
    AF_uint32   IOMemoryLen[4];     /* Length of I/O memory maps        */
    AF_uint32   res1[10];           /* Reserved for future expansion    */

    /*-------------------------------------------------------------------*/
    /* Near pointers mapped by application for driver                    */
    /*-------------------------------------------------------------------*/

    void        *IOMemMaps[4];      /* Pointers to mapped I/O memory    */
    void        *BankedMem;         /* Ptr to mapped banked video mem   */
    void        *LinearMem;         /* Ptr to mapped linear video mem   */

    /*-------------------------------------------------------------------*/
    /* Important selectors allocated by application for driver           */
    /*-------------------------------------------------------------------*/

    AF_uint32   Sel0000h;           /* 1Mb selector to entire first Mb  */
    AF_uint32   Sel0040h;           /* Selector to segment at 0x0040:0  */
    AF_uint32   SelA000h;           /* Selector to segment at 0xA000:0  */
    AF_uint32   SelB000h;           /* Selector to segment at 0xB000:0  */
    AF_uint32   SelC000h;           /* Selector to segment at 0xC000:0  */

    /*-------------------------------------------------------------------*/
    /* Device driver state variables                                     */
    /*-------------------------------------------------------------------*/

    AF_uint32   BufferEndX;         /* Last X coord of each buffer      */
    AF_uint32   BufferEndY;         /* Last Y coord of each buffer      */
```

```
    AF_uint32   OriginOffset;       /* Current start of active page     */
    AF_uint32   OffscreenOffset;    /* Start of offscreen memory area   */
    AF_uint32   OffscreenStartY;    /* First Y coord of offscreen mem   */
    AF_uint32   OffscreenEndY;      /* Last Y coord of offscreen mem    */
    AF_uint32   res2[10];           /* Reserved for future expansion    */

    /*------------------------------------------------------------------*/
    /* Relocateable 32 bit bank switch routine, needed for framebuffer  */
    /* virtualisation under Windows with DVA.386/VFLATD.386. This       */
    /* function *MUST* program the bank with IO mapped registers, as    */
    /* when the function is called there is no way to provide access to */
    /* the devices memory mapped registers (because there is no way to  */
    /* for it to gain access to a copy of this AF_devCtx block). For    */
    /* devices that only have memory mapped registers, this vector      */
    /* *MUST* be NULL indicating that this is not supported. However    */
    /* all these devices all have a real linear framebuffer anyway,     */
    /* so the virtualisation services will not be needed.               */
    /*------------------------------------------------------------------*/

    AF_uint32   SetBank32Len;           /* Length of 32 bit code        */
    void        *SetBank32;             /* 32 bit relocateable code     */

    /*------------------------------------------------------------------*/
    /* REQUIRED callback functions provided by application              */
    /*------------------------------------------------------------------*/

    void        *Int86;                 /* Issue real mode interrupt    */
    void        *CallRealMode;          /* Call a real mode function    */

    /*------------------------------------------------------------------*/
    /* Device driver functions                                          */
    /*------------------------------------------------------------------*/

    void        *InitDriver;            /* Initialise driver            */
    void        *GetVideoModeInfo;      /* Get video mode information   */
    void        *SetVideoMode;          /* Set a video mode             */
    void        *RestoreTextMode;       /* Restore text mode operation  */
    void        *SetBank;               /* Set framebuffer bank         */
    void        *SetDisplayStart;       /* Set virtual display start    */
    void        *SetActiveBuffer;       /* Set active output buffer     */
    void        *SetVisibleBuffer;      /* Set Visible display buffer   */
    void        *SetPaletteData;        /* Program palette data         */
    void        *SetGammaCorrectData;   /* Program gamma correct'n data */
    void        *WaitTillIdle;          /* Wait till engine is idle     */
    void        *EnableDirectAccess;    /* Enable direct mem access     */
    void        *DisableDirectAccess;   /* Disable direct mem access    */
    void        *SetCursor;             /* Download hardware cursor     */
    void        *SetCursorPos;          /* Set cursor position          */
    void        *SetCursorColor;        /* Set cursor color             */
    void        *ShowCursor;            /* Show/hide cursor             */
    void        *SetMix;                /* Set ALU mix operations       */
    void        *Set8x8MonoPattern;     /* Set 8x8 mono bitmap pattern  */
    void        *Set8x8ColorPattern;    /* Set 8x8 color bitmap pattern */
    void        *SetLineStipple;        /* Set 16 bit line stipple      */
    void        *SetClipRect;           /* Set clipping rectangle       */
    void        *DrawScan;              /* Draw a solid scanline        */
    void        *DrawPattScan;          /* Draw a patterned scanline    */
    void        *DrawColorPattScan;     /* Draw color pattern scanline  */
    void        *DrawScanList;          /* Draw list of solid scanlines */
    void        *DrawRect;              /* Draw a solid rectangle       */
    void        *DrawPattRect;          /* Draw a patterned rectangle   */
    void        *DrawColorPattRect;     /* Draw color pattern rectangle */
    void        *DrawLine;              /* Draw a solid line            */
    void        *DrawStippleLine;       /* Draw a stippled line         */
    void        *DrawTrap;              /* Draw a solid trapezoid       */
    void        *DrawTri;               /* Draw a solid triangle        */
    void        *DrawQuad;              /* Draw a solid quad            */
    void        *PutMonoImage;          /* Display a monochrome bitmap  */
    void        *BitBlt;                /* Blt screen to screen         */
    void        *BitBltLin;             /* Linear source BitBlt         */
    void        *SrcTransBlt;           /* Source transparent BitBlt    */
    void        *SrcTransBltLin;        /* Linear source SrcTransBlt    */
    void        *DstTransBlt;           /* Dest. transparent BitBlt     */
    void        *DstTransBltLin;        /* Linear source DstTransBlt    */
    } AF_devCtx;
```

```
/* Video mode information block */

typedef struct {
    AF_uint16   Attributes;         /* Mode attributes                  */
    AF_uint16   XResolution;        /* Horizontal resolution in pixels  */
    AF_uint16   YResolution;        /* Vertical resolution in pixels    */
    AF_uint16   BytesPerScanLine;   /* Bytes per horizontal scan line   */
    AF_uint16   BitsPerPixel;       /* Bits per pixel                   */
    AF_uint16   MaxBuffers;         /* Maximum num. of display buffers  */

    /* RGB pixel format info */
    AF_uint8    RedMaskSize;        /* Size of direct color red mask    */
    AF_uint8    RedFieldPosition;   /* Bit posn of lsb of red mask      */
    AF_uint8    GreenMaskSize;      /* Size of direct color green mask  */
    AF_uint8    GreenFieldPosition; /* Bit posn of lsb of green mask    */
    AF_uint8    BlueMaskSize;       /* Size of direct color blue mask   */
    AF_uint8    BlueFieldPosition;  /* Bit posn of lsb of blue mask     */
    AF_uint8    RsvdMaskSize;       /* Size of direct color res mask    */
    AF_uint8    RsvdFieldPosition;  /* Bit posn of lsb of res mask      */

    /* Virtual buffer dimensions */
    AF_uint16   MaxBytesPerScanLine;/* Maximum bytes per scan line      */
    AF_uint16   MaxScanLineWidth;   /* Maximum pixels per scan line     */
    AF_uint8    reserved[118];      /* Pad to 128 byte block size       */
    } AF_modeInfo;

#define VBEAF_DRV       "VBEAF.DRV" /* Name of driver file on disk      */
#define VBEAF_PATH      "VBEAF_PATH"/* Name of environment variable     */
#define VBEAF_VERSION   0x100       /* Lowest version we can work with  */

/* Flags for combining with video modes during mode set */

#define afDontClear     0x8000      /* Dont clear display memory        */
#define afLinearBuffer  0x4000      /* Enable linear framebuffer mode   */
#define afMultiBuffer   0x2000      /* Enable multi buffered mode       */
#define afVirtualScroll 0x1000      /* Enable virtual scrolling         */

/* Flags for the mode attributes returned by GetModeInfo */

#define afHaveMultiBuffer   0x0001  /* Mode supports multi buffering    */
#define afHaveVirtualScroll 0x0002  /* Mode supports virtual scrolling  */
#define afHaveBankedBuffer  0x0004  /* Mode supports banked framebuffer */
#define afHaveLinearBuffer  0x0008  /* Mode supports linear framebuffer */
#define afHaveAccel2D       0x0010  /* Mode supports 2D acceleration    */
#define afHaveDualBuffers   0x0020  /* Mode uses dual buffers           */
#define afHaveHWCursor      0x0040  /* Mode supports a hardware cursor  */
#define afHave8BitDAC       0x0080  /* Mode uses an 8 bit palette DAC   */
#define afNonVGAMode        0x0100  /* Mode is a NonVGA mode            */

/* Types of mix operations supported */

typedef enum {                      /* Write mode operators             */
    AF_REPLACE_MIX,                 /* Replace mode                     */
    AF_AND_MIX,                     /* AND mode                         */
    AF_OR_MIX,                      /* OR mode                          */
    AF_XOR_MIX,                     /* XOR mode                         */
    AF_NOP_MIX,                     /* Destination pixel unchanged      */
    } AF_mixModes;

/* Palette entry structure, always in 8 bits per primary format */

typedef struct {
    AF_uint8    blue;               /* Blue component of color          */
    AF_uint8    green;              /* Green component of color         */
    AF_uint8    red;                /* Blue component of color          */
    AF_uint8    alpha;              /* Alpha or alignment byte          */
    } AF_palette;

/* Hardware cursor structure */

typedef struct {
    AF_uint32   xorMask[32];        /* Cursor XOR mask                  */
    AF_uint32   andMask[32];        /* Cursor AND mask                  */
    AF_uint32   hotx;               /* Cursor X coordinate hot spot     */
```

```
    AF_uint32   hoty;               /* Cursor Y coordinate hot spot     */
    } AF_cursor;

/* Integer coordinates passed to DrawLineList */

typedef struct {
    int x;
    int y;
    } AF_point;

/* 16.16 fixed point coordinates passed for triangle and quad fills */

typedef struct {
    AF_fix32    x;
    AF_fix32    y;
    } AF_fxpoint;

/* Macros to convert between integer and 32 bit fixed point format */

#define AF_FIX_1            0x10000L
#define AF_FIX_2            0x20000L
#define AF_FIX_HALF         0x08000L
#define AF_TOFIX(i)         ((long)(i) << 16)
#define AF_FIXTOINT(f)      ((int)((f) >> 16))
#define AF_FIXROUND(f)      ((int)(((f) + MGL_FIX_HALF) >> 16))

/* DPMI register structure used in calls to Int86 and CallRealMode */

typedef struct {
    long    edi;
    long    esi;
    long    ebp;
    long    reserved;
    long    ebx;
    long    edx;
    long    ecx;
    long    eax;
    short   flags;
    short   es,ds,fs,gs,ip,cs,sp,ss;
    } AF_DPMI_regs;

/* Flags returned by AF_status to indicate driver load status */

typedef enum {
    afOK,               /* No error                                     */
    afNotDetected,      /* Graphics hardware not detected               */
    afDriverNotFound,   /* Driver file not found                        */
    afCorruptDriver,    /* File loaded not a driver file                */
    afLoadMem,          /* Not enough memory to load driver             */
    afOldVersion,       /* Driver file is an older version              */
    afMemMapError,      /* Could not map physical memory areas          */
    afMaxError,
    } AF_errorType;

/* Default locations to find the driver for different operating systems */

#define AF_DRIVERDIR_DOS    "c:\\"      /* DOS, Windows and OS/2        */
#define AF_DRIVERDIR_UNIX   "/"         /* Unix                         */

#pragma pack()

/*------------------------- Function Prototypes --------------------------*/

#ifdef  __cplusplus
extern "C" {                        /* Use "C" linkage when in C++ mode */
#endif

/* Function to load the VBEAF.DRV driver file and initialise it */

AF_devCtx   * AFAPI AF_loadDriver(const char *driverDir);
void        AFAPI AF_unloadDriver(AF_devCtx *drv);
AF_int32    AFAPI AF_status(void);
const char  * AFAPI AF_errorMsg(int status);

/* The following provides a high level C based API to the accelerated
```

```
 * rendering functions. For maximum performance, you should make direct
 * calls to the accelerated rendering functions in assembler from your own
 * rendering routines.
 */

AF_int32    AFAPI AF_getVideoModeInfo(AF_devCtx *dc,AF_int16 mode,AF_modeInfo *modeInfo);
AF_int32    AFAPI AF_setVideoMode(AF_devCtx *dc,AF_int16 mode,AF_int32 *bytesPerLine,int
numBuffers);
AF_int32    AFAPI AF_setVirtualVideoMode(AF_devCtx *dc,AF_int16 mode,AF_int32 virtualX,AF_int32
virtualY,AF_int32 *bytesPerLine,int numBuffers);
void        AFAPI AF_restoreTextMode(AF_devCtx *dc);
void        AFAPI AF_setDisplayStart(AF_devCtx *dc,AF_int32 x,AF_int32 y,AF_int32 waitVRT);
void        AFAPI AF_setActiveBuffer(AF_devCtx *dc,AF_int32 index);
void        AFAPI AF_setVisibleBuffer(AF_devCtx *dc,AF_int32 index,AF_int32 waitVRT);
void        AFAPI AF_setPaletteData(AF_devCtx *dc,AF_palette *pal,AF_int32 num,AF_int32
index,AF_int32 waitVRT);
void        AFAPI AF_setGammaCorrectData(AF_devCtx *dc,AF_palette *pal,AF_int32 num,AF_int32
index);
void        AFAPI AF_setBank(AF_devCtx *dc,AF_int32 bank);
void        AFAPI AF_waitTillIdle(AF_devCtx *dc);
void        AFAPI AF_enableDirectAccess(AF_devCtx *dc);
void        AFAPI AF_disableDirectAccess(AF_devCtx *dc);
void        AFAPI AF_setCursor(AF_devCtx *dc,AF_cursor *cursor);
void        AFAPI AF_setCursorPos(AF_devCtx *dc,AF_int32 x,AF_int32 y);
void        AFAPI AF_setCursorColor(AF_devCtx *dc,AF_uint8 red,AF_uint8 green,AF_uint8 blue);
void        AFAPI AF_showCursor(AF_devCtx *dc,AF_int32 visible);
void        AFAPI AF_setMix(AF_devCtx *dc,AF_int32 foreMix,AF_int32 backMix);
void        AFAPI AF_set8x8MonoPattern(AF_devCtx *dc,AF_pattern *pattern);
void        AFAPI AF_setLineStipple(AF_devCtx *dc,AF_stipple stipple);
void        AFAPI AF_setClipRect(AF_devCtx *dc,AF_int32 minx,AF_int32 miny,AF_int32 maxx,AF_int32
maxy);
void        AFAPI AF_drawScan(AF_devCtx *dc,AF_int32 color,AF_int32 y,AF_int32 x1,AF_int32 x2);
void        AFAPI AF_drawPattScan(AF_devCtx *dc,AF_int32 foreColor,AF_int32 backColor,AF_int32
y,AF_int32 x1,AF_int32 x2);
void        AFAPI AF_drawScanList(AF_devCtx *dc,AF_color color,AF_int32 y,AF_int32
length,AF_int16 *scans);
void        AFAPI AF_drawRect(AF_devCtx *dc,AF_color color,AF_int32 left,AF_int32 top,AF_int32
width,AF_int32 height);
void        AFAPI AF_drawPattRect(AF_devCtx *dc,AF_color foreColor,AF_color backColor,AF_int32
left,AF_int32 top,AF_int32 width,AF_int32 height);
void        AFAPI AF_drawLine(AF_devCtx *dc,AF_color color,AF_fix32 x1,AF_fix32 y1,AF_fix32
x2,AF_fix32 y2);
void        AFAPI AF_drawStippleLine(AF_devCtx *dc,AF_color foreColor,AF_color backColor,AF_fix32
x1,AF_fix32 y1,AF_fix32 x2,AF_fix32 y2);
void        AFAPI AF_drawTrap(AF_devCtx *dc,AF_color color,AF_int32 y,AF_int32 count,AF_fix32
x1,AF_fix32 x2,AF_fix32 slope1,AF_fix32 slope2);
void        AFAPI AF_drawTri(AF_devCtx *dc,AF_color color,AF_fxpoint *v1,AF_fxpoint
*v2,AF_fxpoint *v3,AF_fix32 xOffset,AF_fix32 yOffset);
void        AFAPI AF_drawQuad(AF_devCtx *dc,AF_color color,AF_fxpoint *v1,AF_fxpoint
*v2,AF_fxpoint *v3,AF_fxpoint *v4,AF_fix32 xOffset,AF_fix32 yOffset);
void        AFAPI AF_putMonoImage(AF_devCtx *dc,AF_int32 foreColor,AF_int32 backColor,AF_int32
x,AF_int32 y,AF_int32 byteWidth,AF_int32 height,AF_uint8 *image);
void        AFAPI AF_bitBlt(AF_devCtx *dc,AF_int32 left,AF_int32 top,AF_int32 width,AF_int32
height,AF_int32 dstLeft,AF_int32 dstTop,AF_int32 op);
void        AFAPI AF_bitBltLin(AF_devCtx *dc,AF_int32 srcOfs,AF_int32 dstLeft,AF_int32
dstTop,AF_int32 width,AF_int32 height,AF_int32 op);
void        AFAPI AF_srcTransBlt(AF_devCtx *dc,AF_int32 left,AF_int32 top,AF_int32 width,AF_int32
height,AF_int32 dstLeft,AF_int32 dstTop,AF_int32 op,AF_color transparent);
void        AFAPI AF_srcTransBltLin(AF_devCtx *dc,AF_int32 srcOfs,AF_int32 dstLeft,AF_int32
dstTop,AF_int32 width,AF_int32 height,AF_int32 op,AF_color transparent);
void        AFAPI AF_dstTransBlt(AF_devCtx *dc,AF_int32 left,AF_int32 top,AF_int32 width,AF_int32
height,AF_int32 dstLeft,AF_int32 dstTop,AF_int32 op,AF_color transparent);
void        AFAPI AF_dstTransBltLin(AF_devCtx *dc,AF_int32 srcOfs,AF_int32 dstLeft,AF_int32
dstTop,AF_int32 width,AF_int32 height,AF_int32 op,AF_color transparent);

#ifdef  __cplusplus
}                               /* End of "C" linkage for C++       */
#endif

#endif  /* __VBEAF_H */
```

VBEAF.C

```
/****************************************************************************
*
*                VESA BIOS Extensions/Accelerator Functions
*                            Version 1.0
*
*                   Copyright (C) 1996 SciTech Software.
*                           All rights reserved.
*
* Filename:     $Workfile:   vbeaf.c  $
* Developed by: SciTech Software
*
* Language:     ANSI C
* Environment:  IBM PC 32 bit Protected Mode.
*
* Description:  C module for the Graphics Acclerator Driver API. Uses
*               the SciTech PM/Pro library for interfacing with DOS
*               extender specific functions.
*
* $Date:   17 Feb 1996 19:34:46  $ $Author:   KendallB  $
*
****************************************************************************/

#ifdef  MGLWIN
#include "mgl.h"
#endif
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "vbeaf.h"
#include "pmode.h"

#if !defined(__16BIT__) || defined(TESTING)

/*------------------------- Global Variables ------------------------------*/

#define AF_DRIVERDIR    AF_DRIVERDIR_DOS

static int          status = afOK;
static void         *IOMemMaps[4] = {NULL,NULL,NULL,NULL};
static void         *BankedMem = NULL;
static void         *LinearMem = NULL;
static AF_int32     Sel0000h = 0;
static AF_int32     Sel0040h = 0;
static AF_int32     SelA000h = 0;
static AF_int32     SelB000h = 0;
static AF_int32     SelC000h = 0;

/*------------------------- Implementation --------------------------------*/

/* Internal assembler functions */

AF_int32    _cdecl _AF_initDriver(AF_devCtx *dc);
void        _cdecl _AF_int86(void);
void        _cdecl _AF_callRealMode(void);

static void backslash(char *s)
/****************************************************************************
*
* Function:     backslash
* Parameters:   s   - String to add backslash to
*
* Description:  Appends a trailing '\' pathname separator if the string
*               currently does not have one appended.
*
****************************************************************************/
{
    uint pos = strlen(s);
    if (s[pos-1] != '\\') {
        s[pos] = '\\';
        s[pos+1] = '\0';
        }
}
```

```
static long fileSize(FILE *f)
/****************************************************************************
*
* Function:     fileSize
* Parameters:   f   - Open file to determine the size of
* Returns:      Length of the file in bytes.
*
* Description:  Determines the length of the file, without altering the
*               current position in the file.
*
****************************************************************************/
{
    long    size,oldpos = ftell(f);

    fseek(f,0,SEEK_END);            /* Seek to end of file              */
    size = ftell(f);                /* Determine the size of the file   */
    fseek(f,oldpos,SEEK_SET);       /* Seek to old position in file     */
    return size;                    /* Return the size of the file      */
}

void _cdecl _AF_callRealMode_C(AF_DPMI_regs *dregs)
/****************************************************************************
*
* Function:     _AF_callRealMode_C
* Parameters:   dregs   - Pointer to DPMI register structure
*
* Description:  Calls a real mode procedure. This does not need to be
*               speedy, so we simply convert the registers to the format
*               expected by the PM/Pro library and let it handle it.
*
****************************************************************************/
{
    RMREGS  regs;
    RMSREGS sregs;

    regs.x.ax = (short)dregs->eax;
    regs.x.bx = (short)dregs->ebx;
    regs.x.cx = (short)dregs->ecx;
    regs.x.dx = (short)dregs->edx;
    regs.x.si = (short)dregs->esi;
    regs.x.di = (short)dregs->edi;
    sregs.es = dregs->es;
    sregs.ds = dregs->ds;

    PM_callRealMode(dregs->cs,dregs->ip,&regs,&sregs);

    dregs->eax = regs.x.ax;
    dregs->ebx = regs.x.bx;
    dregs->ecx = regs.x.cx;
    dregs->edx = regs.x.dx;
    dregs->esi = regs.x.si;
    dregs->edi = regs.x.di;
    dregs->es = sregs.es;
    dregs->ds = sregs.ds;
}

void _cdecl _AF_int86_C(AF_int32 intno,AF_DPMI_regs *dregs)
/****************************************************************************
*
* Function:     _AF_int86_C
* Parameters:   intno   - Interrupt number to issue
*               dregs   - Pointer to DPMI register structure
*
* Description:  Issues a real mode interrupt. This does not need to be
*               speedy, so we simply convert the registers to the format
*               expected by the PM/Pro library and let it handle it.
*
****************************************************************************/
{
    RMREGS  regs;
    RMSREGS sregs;

    regs.x.ax = (short)dregs->eax;
    regs.x.bx = (short)dregs->ebx;
```

```
    regs.x.cx = (short)dregs->ecx;
    regs.x.dx = (short)dregs->edx;
    regs.x.si = (short)dregs->esi;
    regs.x.di = (short)dregs->edi;
    sregs.es = dregs->es;
    sregs.ds = dregs->ds;

    PM_int86x(intno,&regs,&regs,&sregs);

    dregs->eax = regs.x.ax;
    dregs->ebx = regs.x.bx;
    dregs->ecx = regs.x.cx;
    dregs->edx = regs.x.dx;
    dregs->esi = regs.x.si;
    dregs->edi = regs.x.di;
    dregs->es = sregs.es;
    dregs->ds = sregs.ds;
}

AF_int32 _AF_initInternal(AF_devCtx *dc)
/****************************************************************************
*
* Function:     _AF_initInternal
* Parameters:   dc  - Pointer to device context
* Returns:      Error code.
*
* Description:  Performs internal initialisation on the AF_devCtx driver
*               block, assuming that it has been loaded correctly.
*
****************************************************************************/
{
    /* Verify that the file is the driver file we are expecting */
    if (strcmp(dc->Signature,VBEAF_DRV) != 0)
        return status = afCorruptDriver;
    if (dc->Version < VBEAF_VERSION)
        return status = afOldVersion;

    /* Map the memory mapped register locations for the driver. We need to
     * map up to four different locations that may possibly be needed. If
     * the base address is zero then the memory does not need to be mapped.
     */
    if (IOMemMaps[0] == NULL) {
        if (dc->IOMemoryBase[0]) {
            IOMemMaps[0] = PM_mapPhysicalAddr(dc->IOMemoryBase[0],dc->IOMemoryLen[0]-1);
            if (IOMemMaps[0] == NULL)
                return status = afMemMapError;
            }
        if (dc->IOMemoryBase[1]) {
            IOMemMaps[1] = PM_mapPhysicalAddr(dc->IOMemoryBase[1],dc->IOMemoryLen[1]-1);
            if (IOMemMaps[1] == NULL)
                return status = afMemMapError;
            }
        if (dc->IOMemoryBase[2]) {
            IOMemMaps[2] = PM_mapPhysicalAddr(dc->IOMemoryBase[2],dc->IOMemoryLen[2]-1);
            if (IOMemMaps[2] == NULL)
                return status = afMemMapError;
            }
        if (dc->IOMemoryBase[3]) {
            IOMemMaps[3] = PM_mapPhysicalAddr(dc->IOMemoryBase[3],dc->IOMemoryLen[3]-1);
            if (IOMemMaps[3] == NULL)
                return status = afMemMapError;
            }
        }
    dc->IOMemMaps[0] = IOMemMaps[0];
    dc->IOMemMaps[1] = IOMemMaps[1];
    dc->IOMemMaps[2] = IOMemMaps[2];
    dc->IOMemMaps[3] = IOMemMaps[3];

    /* Map the banked video memory area for the driver */
    if (BankedMem == NULL && dc->BankedBasePtr) {
        BankedMem = PM_mapPhysicalAddr(dc->BankedBasePtr,0xFFFF);
        if (BankedMem == NULL)
            return status = afMemMapError;
        }
    dc->BankedMem = BankedMem;
```

```
    /* Map the linear video memory area for the driver */
    if (LinearMem == NULL && dc->LinearBasePtr) {
        LinearMem = PM_mapPhysicalAddr(dc->LinearBasePtr,dc->LinearSize*1024L - 1);
        if (LinearMem == NULL)
            return status = afMemMapError;
        }
    dc->LinearMem = LinearMem;

    /* Provide selectors to important real mode segment areas */
    if (Sel0000h == 0) {
        Sel0000h = PM_createSelector(0x00000L,0xFFFFFL);
        Sel0040h = PM_createSelector(0x00400L,0xFFFF);
        SelA000h = PM_createSelector(0xA0000L,0xFFFF);
        SelB000h = PM_createSelector(0xB0000L,0xFFFF);
        SelC000h = PM_createSelector(0xC0000L,0xFFFF);
        }
    dc->Sel0000h = Sel0000h;
    dc->Sel0040h = Sel0040h;
    dc->SelA000h = SelA000h;
    dc->SelB000h = SelB000h;
    dc->SelC000h = SelC000h;

    /* Install the device callback functions */
    dc->Int86 = _AF_int86;
    dc->CallRealMode = _AF_callRealMode;
    return afOK;
}

AF_devCtx * AFAPI AF_loadDriver(const char *driverDir)
/****************************************************************************
*
* Function:     AF_loadDriver
* Parameters:   driverDir   - Directory to load the driver file from
* Returns:      Pointer to the loaded driver file.
*
* Description:  Loads the driver file and intialises the device context
*               ready for use. If the driver file cannot be found, or the
*               driver does not detect the installed hardware, we return
*               NULL and the application can get the status code with
*               AF_status().
*
****************************************************************************/
{
    char        filename[_MAX_PATH];
    FILE        *f;
    int         size;
    AF_devCtx   *dc;

    /* Reset status flag */
    status = afOK;

    /* Try if the default operating system location first */
    strcpy(filename,AF_DRIVERDIR);
    strcat(filename,VBEAF_DRV);
    if ((f = fopen(filename,"rb")) == NULL) {
        /* Now try to find the driver in the VBEAF_PATH directory */
        if (getenv(VBEAF_PATH)) {
            strcpy(filename, getenv(VBEAF_PATH));
            backslash(filename);
            strcat(filename,VBEAF_DRV);
            if ((f = fopen(filename,"rb")) != NULL)
                goto FoundDriver;
            }

        /* Else try in the specified path */
        if (driverDir) {
            strcpy(filename, driverDir);
            backslash(filename);
            strcat(filename,VBEAF_DRV);
            if ((f = fopen(filename,"rb")) != NULL)
                goto FoundDriver;
            }

        /* Driver file was not found */
```

```
        status = afDriverNotFound;
        return NULL;
        }

    /* Allocate memory for driver file and load it */
FoundDriver:
    size = fileSize(f);
    if ((dc = malloc(size+16)) == NULL) {
        status = afLoadMem;
        fclose(f);
        return NULL;
        }
    fread(dc,1,size,f);
    fclose(f);

    /* Perform internal initialisation */
    if (_AF_initInternal(dc) != afOK)
        goto Error;

    /* Now call the driver to detect the installed hardware and initialise
     * the driver.
     */
    if (_AF_initDriver(dc) != 0) {
        status = afNotDetected;
        goto Error;
        }
    return dc;

Error:
    free(dc);
    return NULL;
}

void AFAPI AF_unloadDriver(AF_devCtx *dc)
/****************************************************************************
*
* Function:     AF_unloadDriver
* Parameters:   dc  - Pointer to device context
*
* Description:  Unloads the loaded device driver.
*
****************************************************************************/
{
    free(dc);
}

AF_int32 AFAPI AF_status(void)
{ return status; }

const char * AFAPI AF_errorMsg(int status)
/****************************************************************************
*
* Function:     AF_errorMsg
* Returns:      String describing error condition.
*
****************************************************************************/
{
    const char *msg[] = {
        "No error",
        "Graphics hardware not detected",
        "Driver file not found",
        "File loaded was not a driver file",
        "Not enough memory to load driver",
        "Driver file is an older version",
        "Could not map physical memory areas",
        };
    if (status >= afOK && status < afMaxError)
        return msg[status];
    return "Unknown error!";
}

#endif  /* !defined(__16BIT__) */
```

VBEAF.INC

```
;****************************************************************************
;*
;*               VESA BIOS Extensions/Accelerator Functions
;*                            Version 1.0
;*
;*                   Copyright (C) 1996 SciTech Software.
;*                           All rights reserved.
;*
;* Filename:     $Workfile:   vbeaf.inc  $
;* Developed by: SciTech Software
;*
;* Language:     80386 Assembler (TASM ideal mode)
;* Environment:  IBM PC 32 bit Protected Mode.
;*
;* Description:  Macros and type definitions for VBE/AF
;*
;* $Date:   21 Feb 1996 18:35:42  $ $Author:   KendallB  $
;*
;****************************************************************************

;----------------------------------------------------------------------------
; Fundamental types
;----------------------------------------------------------------------------

typedef AF_int8     byte
typedef AF_int16    word
typedef AF_int32    dword
typedef AF_uint8    byte
typedef AF_uint16   word
typedef AF_uint32   dword
typedef AF_fix32    dword

typedef AF_pattern  AF_uint8
typedef AF_stipple  AF_uint16
typedef AF_color    AF_uint32

true                =   1
false               =   0

;----------------------------------------------------------------------------
; Attribute flags
;----------------------------------------------------------------------------

afDontClear         =   8000h
afLinearBuffer      =   4000h
afMultiBuffer       =   2000h
afVirtualScroll     =   1000h

afHaveMultiBuffer   =   0001h
afHaveVirtualScroll =   0002h
afHaveBankedBuffer  =   0004h
afHaveLinearBuffer  =   0008h
afHaveAccel2D       =   0010h
afHaveDualBuffers   =   0020h
afHaveHWCursor      =   0040h
afHave8BitDAC       =   0080h
afNonVGAMode        =   0100h

enum AF_mixmodes {
    AF_REPLACE_MIX
    AF_AND_MIX
    AF_OR_MIX
    AF_XOR_MIX
    AF_NOP_MIX
    }

;----------------------------------------------------------------------------
; Public device context structure
;----------------------------------------------------------------------------

struc   AF_devCtx_s
Signature               uchar 12 dup (?)
```

```
AFVersion               AF_uint32   ?
DriverRev               AF_uint32   ?
OemVendorName           uchar 80 dup (?)
OemCopyright            uchar 80 dup (?)
AvailableModes          dptr        ?
TotalMemory             AF_uint32   ?
Attributes              AF_uint32   ?
BankSize                AF_uint32   ?
BankedBasePtr           AF_uint32   ?
LinearSize              AF_uint32   ?
LinearBasePtr           AF_uint32   ?
LinearGranularity       AF_uint32   ?
IOPortsTable            dptr        ?
IOMemoryBase            AF_uint32 4 dup (?)
IOMemoryLen             AF_uint32 4 dup (?)
res1                    AF_uint32 10 dup (?)
IOMemMaps               dptr 4 dup (?)
BankedMem               dptr        ?
LinearMem               dptr        ?
Sel0000h                AF_uint16   ?
pad1                    AF_uint16   ?
Sel0040h                AF_uint16   ?
pad2                    AF_uint16   ?
SelA000h                AF_uint16   ?
pad3                    AF_uint16   ?
SelB000h                AF_uint16   ?
pad4                    AF_uint16   ?
SelC000h                AF_uint16   ?
pad5                    AF_uint16   ?
BufferEndX              AF_uint32   ?
BufferEndY              AF_uint32   ?
OriginOffset            AF_uint32   ?
OffscreenOffset         AF_uint32   ?
OffscreenStartY         AF_uint32   ?
OffscreenEndY           AF_uint32   ?
res2                    AF_uint32 10 dup (?)
SetBank32Len            AF_uint32   ?
SetBank32               cptr        ?
Int86                   cptr        ?
CallRealMode            cptr        ?
InitDriver              cptr        ?
GetVideoModeInfo        cptr        ?
SetVideoMode            cptr        ?
RestoreTextMode         cptr        ?
SetBank                 cptr        ?
SetDisplayStart         cptr        ?
SetActiveBuffer         cptr        ?
SetVisibleBuffer        cptr        ?
SetPaletteData          cptr        ?
SetGammaCorrectData     cptr        ?
WaitTillIdle            cptr        ?
EnableDirectAccess      cptr        ?
DisableDirectAccess     cptr        ?
SetCursor               cptr        ?
SetCursorPos            cptr        ?
SetCursorColor          cptr        ?
ShowCursor              cptr        ?
SetMix                  cptr        ?
Set8x8MonoPattern       cptr        ?
Set8x8ColorPattern      cptr        ?
SetLineStipple          cptr        ?
SetClipRect             cptr        ?
DrawScan                cptr        ?
DrawPattScan            cptr        ?
DrawColorPattScan       cptr        ?
DrawScanList            cptr        ?
DrawRect                cptr        ?
DrawPattRect            cptr        ?
DrawColorPattRect       cptr        ?
DrawLine                cptr        ?
DrawStippleLine         cptr        ?
DrawTrap                cptr        ?
DrawTri                 cptr        ?
DrawQuad                cptr        ?
PutMonoImage            cptr        ?
```

```
BitBlt                  cptr        ?
BitBltLin               cptr        ?
SrcTransBlt             cptr        ?
SrcTransBltLin          cptr        ?
DstTransBlt             cptr        ?
DstTransBltLin          cptr        ?
ends    AF_devCtx_s

FIRST_AF_VEC    EQU InitDriver
LAST_AF_VEC     EQU DstTransBltLin

AF_devCtx = (AF_devCtx_s PTR DS:EBX)

;----------------------------------------------------------------------------
; Mode information block structure
;----------------------------------------------------------------------------

struc   AF_modeInfo_s
Attributes              AF_uint16   ?
XResolution             AF_uint16   ?
YResolution             AF_uint16   ?
BytesPerScanLine        AF_uint16   ?
BitsPerPixel            AF_uint16   ?
MaxBuffers              AF_uint16   ?
RedMaskSize             AF_uint8    ?
RedFieldPosition        AF_uint8    ?
GreenMaskSize           AF_uint8    ?
GreenFieldPosition      AF_uint8    ?
BlueMaskSize            AF_uint8    ?
BlueFieldPosition       AF_uint8    ?
RsvdMaskSize            AF_uint8    ?
RsvdFieldPosition       AF_uint8    ?
MaxBytesPerScanLine     AF_uint16   ?
MaxScanLineWidth        AF_uint16   ?
reserved                uchar 118 dup (?)
ends    AF_modeInfo_s

AF_modeInfo = (AF_modeInfo_s PTR DS:EDI)

;----------------------------------------------------------------------------
; DPMI register structure for passing to Int86 and CallRealMode
;----------------------------------------------------------------------------

struc   AF_DPMI_regs_s
edi                     AF_uint32   ?
esi                     AF_uint32   ?
ebp                     AF_uint32   ?
reserved                AF_uint32   ?
ebx                     AF_uint32   ?
edx                     AF_uint32   ?
ecx                     AF_uint32   ?
eax                     AF_uint32   ?
flags                   AF_uint16   ?
es                      AF_uint16   ?
ds                      AF_uint16   ?
fs                      AF_uint16   ?
gs                      AF_uint16   ?
ip                      AF_uint16   ?
cs                      AF_uint16   ?
sp                      AF_uint16   ?
ss                      AF_uint16   ?
ends    AF_DPMI_regs_s

;----------------------------------------------------------------------------
; Palette entry structure
;----------------------------------------------------------------------------

struc   AF_palette_s
blue                    AF_uint8    ?
green                   AF_uint8    ?
red                     AF_uint8    ?
alpha                   AF_uint8    ?
ends    AF_palette_s

AF_palette = (AF_palette_s PTR DS:ESI)
```

```
;----------------------------------------------------------------------------
; Hardware cursor definition structure
;----------------------------------------------------------------------------

struc   AF_cursor_s
xorMask                 AF_int32 32 dup (?)
andMask                 AF_int32 32 dup (?)
hotx                    AF_int32    ?
hoty                    AF_int32    ?
ends    AF_cursor_s

AF_cursor = (AF_cursor_s PTR DS:ESI)

;----------------------------------------------------------------------------
; Parameter block for SetClipRect
;----------------------------------------------------------------------------

struc   AF_clipRect_s
minX                    AF_int32    ?
minY                    AF_int32    ?
maxX                    AF_int32    ?
maxY                    AF_int32    ?
ends    AF_clipRect_s

AF_clipRect = (AF_clipRect_s PTR DS:ESI)

;----------------------------------------------------------------------------
; Parameter block for DrawRect and DrawPattRect
;----------------------------------------------------------------------------

struc   AF_rect_s
left                    AF_int32    ?
top                     AF_int32    ?
width                   AF_int32    ?
height                  AF_int32    ?
ends    AF_rect_s

AF_rect = (AF_rect_s PTR DS:ESI)

;----------------------------------------------------------------------------
; Parameter block for DrawLine and DrawStippleLine
;----------------------------------------------------------------------------

struc   AF_line_s
x1                      AF_fix32    ?
y1                      AF_fix32    ?
x2                      AF_fix32    ?
y2                      AF_fix32    ?
ends    AF_line_s

AF_line = (AF_line_s PTR DS:ESI)

;----------------------------------------------------------------------------
; 2D fixed point vertex structure
;----------------------------------------------------------------------------

struc   AF_fxpoint
x                       AF_fix32    ?
y                       AF_fix32    ?
ends    AF_fxpoint

;----------------------------------------------------------------------------
; Parameter block for DrawTrap
;----------------------------------------------------------------------------

struc   AF_trap_s
y                       AF_uint32   ?
count                   AF_uint32   ?
x1                      AF_fix32    ?
x2                      AF_fix32    ?
slope1                  AF_fix32    ?
slope2                  AF_fix32    ?
ends    AF_trap_s
```

```
AF_trap = (AF_trap_s PTR DS:ESI)

;-----------------------------------------------------------------------------
; Parameter block for DrawTri
;-----------------------------------------------------------------------------

struc   AF_tri_s
v1                      dptr        ?
v2                      dptr        ?
v3                      dptr        ?
xOffset                 AF_fix32    ?
yOffset                 AF_fix32    ?
ends    AF_tri_s

AF_tri = (AF_tri_s PTR DS:ESI)

;-----------------------------------------------------------------------------
; Parameter block for DrawQuad
;-----------------------------------------------------------------------------

struc   AF_quad_s
v1                      dptr        ?
v2                      dptr        ?
v3                      dptr        ?
v4                      dptr        ?
xOffset                 AF_fix32    ?
yOffset                 AF_fix32    ?
ends    AF_quad_s

AF_quad = (AF_quad_s PTR DS:ESI)

;-----------------------------------------------------------------------------
; Parameter block for PutMonoImage
;-----------------------------------------------------------------------------

struc   AF_monoImage_s
x                       AF_int32    ?
y                       AF_int32    ?
byteWidth               AF_int32    ?
height                  AF_int32    ?
image                   dptr        ?
ends    AF_monoImage_s

AF_monoImage = (AF_monoImage_s PTR DS:ESI)

;-----------------------------------------------------------------------------
; Parameter block for BitBlt
;-----------------------------------------------------------------------------

struc   AF_bitBlt_s
left                    AF_int32    ?
top                     AF_int32    ?
width                   AF_int32    ?
height                  AF_int32    ?
dstLeft                 AF_int32    ?
dstTop                  AF_int32    ?
op                      AF_int32    ?
ends    AF_bitBlt_s

AF_bitBlt = (AF_bitBlt_s PTR DS:ESI)

;-----------------------------------------------------------------------------
; Parameter block for BitBltLin
;-----------------------------------------------------------------------------

struc   AF_bitBltLin_s
srcOfs                  AF_int32    ?
dstLeft                 AF_int32    ?
dstTop                  AF_int32    ?
width                   AF_int32    ?
height                  AF_int32    ?
op                      AF_int32    ?
ends    AF_bitBltLin_s

AF_bitBltLin = (AF_bitBltLin_s PTR DS:ESI)
```

```
;----------------------------------------------------------------------------
; Parameter block for TransBlt
;----------------------------------------------------------------------------

struc   AF_transBlt_s
left                    AF_int32    ?
top                     AF_int32    ?
width                   AF_int32    ?
height                  AF_int32    ?
dstLeft                 AF_int32    ?
dstTop                  AF_int32    ?
op                      AF_int32    ?
transparent             AF_int32    ?
ends    AF_transBlt_s

AF_transBlt = (AF_transBlt_s PTR DS:ESI)

;----------------------------------------------------------------------------
; Parameter block for TransBltLin
;----------------------------------------------------------------------------

struc   AF_transBltLin_s
srcOfs                  AF_int32    ?
dstLeft                 AF_int32    ?
dstTop                  AF_int32    ?
width                   AF_int32    ?
height                  AF_int32    ?
op                      AF_int32    ?
transparent             AF_int32    ?
ends    AF_transBltLin_s

AF_transBltLin = (AF_transBltLin_s PTR DS:ESI)
```

_VBEAF.ASM

```
;****************************************************************************
;*
;*               VESA BIOS Extensions/Accelerator Functions
;*                            Version 1.0
;*
;*                   Copyright (C) 1996 SciTech Software.
;*                           All rights reserved.
;*
;* Filename:     $Workfile:   _vbeaf.asm  $
;* Developed by: SciTech Software
;*
;* Language:     80386 Assembler (TASM ideal mode)
;* Environment:  IBM PC 32 bit Protected Mode.
;*
;* Description:  Assembly language support routines for the Graphics
;*               Accelerator API. This module provides a small, sample
;*               C API showing how to call the device context functions
;*               from assembler. For time critical code the device context
;*               functions should be called directly from assembly language
;*               rather than via these C callable functions. However for
;*               many operations these C function will suffice.
;*
;* $Date:   21 Feb 1996 18:35:42  $ $Author:   KendallB  $
;*
;****************************************************************************

        IDEAL

include "model.mac"             ; Memory model macros
include "vbeaf.inc"             ; Structure definitions etc

if flatmodel

header  _vbeaf

        EXTRN   __AF_int86_C:FPTR
        EXTRN   __AF_callRealMode_C:FPTR

begcodeseg  _vbeaf

; Macros to setup and call a generic function that takes a parameter block
; in DS:ESI given the parameters passed on the stack

MACRO   CallGeneric name

        ARG     dc:DPTR, firstParm:UINT

        push    ebp
        mov     ebp,esp
        push    ebx
        push    esi

        mov     ebx,[dc]
        lea     esi,[firstParm]
        call    [AF_devCtx.name]

        pop     esi
        pop     ebx
        pop     ebp
        ret

ENDM

MACRO   CallGeneric1Color name

        ARG     dc:DPTR, color:UINT, firstParm:UINT

        push    ebp
        mov     ebp,esp
        push    ebx
        push    esi
```

```
        mov     eax,[color]
        mov     ebx,[dc]
        lea     esi,[firstParm]
        call    [AF_devCtx.name]

        pop     esi
        pop     ebx
        pop     ebp
        ret

ENDM

MACRO   CallGeneric2Color name

        ARG     dc:DPTR, color:UINT, backColor:UINT, firstParm:UINT

        push    ebp
        mov     ebp,esp
        push    ebx
        push    esi

        mov     eax,[color]
        mov     ebx,[dc]
        mov     ecx,[backColor]
        lea     esi,[firstParm]
        call    [AF_devCtx.name]

        pop     esi
        pop     ebx
        pop     ebp
        ret

ENDM

;----------------------------------------------------------------------------
; _AF_int86     Issue a real mode interrupt
;----------------------------------------------------------------------------
; Entry:        BL      - Interrupt number
;               DS:EDI  - Pointer to DPMI register structure
;
; Exit:         DS:EDI  - Pointer to modifed DPMI register structure
;----------------------------------------------------------------------------
procstart   __AF_int86

        movzx   ebx,bl
        push    edi
        push    ebx
        call    __AF_int86_C            ; Call C version to handle it
        add     esp,8
        ret

procend     __AF_int86

;----------------------------------------------------------------------------
; _AF_callRealMode  Issue a real mode interrupt
;----------------------------------------------------------------------------
; Entry:        BL      - Interrupt number
;               DS:EDI  - Pointer to DPMI register structure
;
; Exit:         DS:EDI  - Pointer to modifed DPMI register structure
;----------------------------------------------------------------------------
procstart   __AF_callRealMode

        push    edi
        call    __AF_callRealMode_C     ; Call C version to handle it
        add     esp,4
        ret

procend     __AF_callRealMode

;----------------------------------------------------------------------------
; AF_int32 _AF_initDriver(AF_devCtx *dc);
;----------------------------------------------------------------------------
procstartdll    __AF_initDriver
```

```
        ARG     dc:DPTR

        push    ebp
        mov     ebp,esp
        push    ebx

        mov     ebx,[dc]
        mov     eax,[AF_devCtx.initDriver]
        add     eax,ebx
        call    eax

        pop     ebx
        pop     ebp
        ret

procend         __AF_initDriver

;----------------------------------------------------------------------------
; AF_int32 AF_getVideoModeInfo(AF_devCtx *dc,AF_int16 mode,
;   AF_modeInfo *modeInfo)
;----------------------------------------------------------------------------
procstartdll    _AF_getVideoModeInfo

        ARG     dc:DPTR, mode:S_USHORT, modeInfo:DPTR

        push    ebp
        mov     ebp,esp
        push    ebx
        push    edi

        movzx   eax,[mode]
        mov     ebx,[dc]
        mov     edi,[modeInfo]
        call    [AF_devCtx.GetVideoModeInfo]

        pop     edi
        pop     ebx
        pop     ebp
        ret

procend         _AF_getVideoModeInfo

;----------------------------------------------------------------------------
; AF_int32 AF_setVideoMode(AF_devCtx *dc,AF_int16 mode,
;   AF_int32 *bytesPerLine,int numBuffers)
;----------------------------------------------------------------------------
procstartdll    _AF_setVideoMode

        ARG     dc:DPTR, mode:S_USHORT, bytesPerLine:DPTR, numBuffers:UINT

        push    ebp
        mov     ebp,esp
        push    ebx

        movzx   eax,[mode]
        mov     ebx,[dc]
        mov     ecx,[bytesPerLine]
        mov     ecx,[ecx]
        mov     edi,[numBuffers]
        call    [AF_devCtx.SetVideoMode]
        mov     edx,[bytesPerLine]
        mov     [edx],ecx

        pop     ebx
        pop     ebp
        ret

procend         _AF_setVideoMode

;----------------------------------------------------------------------------
; AF_int32 AF_setVirtualVideoMode(AF_devCtx *dc,AF_int16 mode,
;   AF_int32 virtualX,AF_int32 virtualY,AF_int32 *bytesPerLine,int numBuffers)
;----------------------------------------------------------------------------
procstartdll    _AF_setVirtualVideoMode
```

```
        ARG     dc:DPTR, mode:S_USHORT, virtualX:UINT, virtualY:UINT,   \
                bytesPerLine:DPTR, numBuffers:UINT

        push    ebp
        mov     ebp,esp
        push    ebx
        push    esi

        movzx   eax,[mode]
        mov     ebx,[dc]
        mov     ecx,[bytesPerLine]
        mov     ecx,[ecx]
        mov     edx,[virtualX]
        mov     esi,[virtualY]
        mov     edi,[numBuffers]
        call    [AF_devCtx.SetVideoMode]
        mov     edx,[bytesPerLine]
        mov     [edx],ecx

        pop     esi
        pop     ebx
        pop     ebp
        ret

procend         _AF_setVirtualVideoMode

;----------------------------------------------------------------------------
; void AF_restoreTextMode(AF_devCtx *dc)
;----------------------------------------------------------------------------
procstartdll    _AF_restoreTextMode

        ARG     dc:DPTR

        push    ebp
        mov     ebp,esp
        push    ebx

        mov     ebx,[dc]
        call    [AF_devCtx.RestoreTextMode]

        pop     ebx
        pop     ebp
        ret

procend         _AF_restoreTextMode

;----------------------------------------------------------------------------
; void AF_setBank(AF_devCtx *dc,AF_int32 bank)
;----------------------------------------------------------------------------
procstartdll    _AF_setBank

        ARG     dc:DPTR, bank:UINT

        push    ebp
        mov     ebp,esp
        push    ebx

        mov     ebx,[dc]
        mov     edx,[bank]
        call    [AF_devCtx.SetBank]

        pop     ebx
        pop     ebp
        ret

procend         _AF_setBank

;----------------------------------------------------------------------------
; void AF_waitTillIdle(AF_devCtx *dc)
;----------------------------------------------------------------------------
procstartdll    _AF_waitTillIdle

        ARG     dc:DPTR

        push    ebp
```

```
        mov     ebp,esp
        push    ebx

        mov     ebx,[dc]
        call    [AF_devCtx.WaitTillIdle]

        pop     ebx
        pop     ebp
        ret

procend         _AF_waitTillIdle

;----------------------------------------------------------------------------
; void AF_enableDirectAccess(AF_devCtx *dc)
;----------------------------------------------------------------------------
procstartdll    _AF_enableDirectAccess

        ARG     dc:DPTR

        push    ebp
        mov     ebp,esp
        push    ebx

        mov     ebx,[dc]
        call    [AF_devCtx.EnableDirectAccess]

        pop     ebx
        pop     ebp
        ret

procend         _AF_enableDirectAccess

;----------------------------------------------------------------------------
; void AF_disableDirectAccess(AF_devCtx *dc)
;----------------------------------------------------------------------------
procstartdll    _AF_disableDirectAccess

        ARG     dc:DPTR

        push    ebp
        mov     ebp,esp
        push    ebx

        mov     ebx,[dc]
        call    [AF_devCtx.DisableDirectAccess]

        pop     ebx
        pop     ebp
        ret

procend         _AF_disableDirectAccess

;----------------------------------------------------------------------------
; void AF_setDisplayStart(AF_devCtx *dc,AF_int32 x,AF_int32 y,
;   AF_int32 waitVRT)
;----------------------------------------------------------------------------
procstartdll    _AF_setDisplayStart

        ARG     dc:DPTR, x:UINT, y:UINT, waitVRT:UINT

        push    ebp
        mov     ebp,esp
        push    ebx

        mov     eax,[waitVRT]
        mov     ebx,[dc]
        mov     ecx,[x]
        mov     edx,[y]
        call    [AF_devCtx.SetDisplayStart]

        pop     ebx
        pop     ebp
        ret

procend         _AF_setDisplayStart
```

```
;----------------------------------------------------------------------------
; void AF_setActiveBuffer(AF_devCtx *dc,AF_int32 index)
;----------------------------------------------------------------------------
procstartdll    _AF_setActiveBuffer

        ARG     dc:DPTR, index:UINT

        push    ebp
        mov     ebp,esp
        push    ebx

        mov     eax,[index]
        mov     ebx,[dc]
        call    [AF_devCtx.SetActiveBuffer]

        pop     ebx
        pop     ebp
        ret

procend         _AF_setActiveBuffer

;----------------------------------------------------------------------------
; void AF_setVisibleBuffer(AF_devCtx *dc,AF_int32 index,AF_int32 waitVRT)
;----------------------------------------------------------------------------
procstartdll    _AF_setVisibleBuffer

        ARG     dc:DPTR, index:UINT, waitVRT:UINT

        push    ebp
        mov     ebp,esp
        push    ebx

        mov     eax,[index]
        mov     ebx,[dc]
        mov     edx,[waitVRT]
        call    [AF_devCtx.SetVisibleBuffer]

        pop     ebx
        pop     ebp
        ret

procend         _AF_setVisibleBuffer

;----------------------------------------------------------------------------
; void AF_setPaletteData(AF_devCtx *dc,AF_palette *pal,AF_int32 num,
;   AF_int32 index,AF_int32 waitVRT)
;----------------------------------------------------------------------------
procstartdll    _AF_setPaletteData

        ARG     dc:DPTR, pal:DPTR, num:UINT, index:UINT, waitVRT:UINT

        push    ebp
        mov     ebp,esp
        push    ebx
        push    esi

        mov     eax,[waitVRT]
        mov     ebx,[dc]
        mov     ecx,[num]
        mov     edx,[index]
        mov     edi,[pal]
        call    [AF_devCtx.SetPaletteData]

        pop     esi
        pop     ebx
        pop     ebp
        ret

procend         _AF_setPaletteData

;----------------------------------------------------------------------------
; void AF_setGammaCorrectData(AF_devCtx *dc,AF_palette *pal,AF_int32 num,
;   AF_int32 index)
;----------------------------------------------------------------------------
```

```
procstartdll    _AF_setGammaCorrectData

        ARG     dc:DPTR, pal:DPTR, num:UINT, index:UINT

        push    ebp
        mov     ebp,esp
        push    ebx
        push    esi

        mov     eax,[index]
        mov     ebx,[dc]
        mov     ecx,[num]
        mov     esi,[pal]
        call    [AF_devCtx.SetGammaCorrectData]

        pop     esi
        pop     ebx
        pop     ebp
        ret

procend         _AF_setGammaCorrectData

;----------------------------------------------------------------------------
; void AF_setCursor(AF_devCtx *dc,AF_cursor *cursor)
;----------------------------------------------------------------------------
procstartdll    _AF_setCursor

        ARG     dc:DPTR, cursor:DPTR

        push    ebp
        mov     ebp,esp
        push    ebx
        push    esi

        mov     ebx,[dc]
        mov     esi,[cursor]
        call    [AF_devCtx.SetCursor]

        pop     esi
        pop     ebx
        pop     ebp
        ret

procend         _AF_setCursor

;----------------------------------------------------------------------------
; void AF_setCursorPos(AF_devCtx *dc,AF_int32 x,AF_int32 y)
;----------------------------------------------------------------------------
procstartdll    _AF_setCursorPos

        ARG     dc:DPTR, x:UINT, y:UINT

        push    ebp
        mov     ebp,esp
        push    ebx

        mov     eax,[x]
        mov     ebx,[dc]
        mov     ecx,[y]
        call    [AF_devCtx.SetCursorPos]

        pop     ebx
        pop     ebp
        ret

procend         _AF_setCursorPos

;----------------------------------------------------------------------------
; void AF_setCursorColor(AF_devCtx *dc,AF_uint8 red,AF_uint8 green,
;   AF_uint8 blue)
;----------------------------------------------------------------------------
procstartdll    _AF_setCursorColor

        ARG     dc:DPTR, red:S_UCHAR, green:S_UCHAR, blue:S_UCHAR
```

```
        push    ebp
        mov     ebp,esp
        push    ebx

        mov     al,[red]
        mov     ah,[green]
        mov     ebx,[dc]
        mov     cl,[blue]
        call    [AF_devCtx.SetCursorColor]

        pop     ebx
        pop     ebp
        ret

procend         _AF_setCursorColor

;----------------------------------------------------------------------------
; void AF_showCursor(AF_devCtx *dc,AF_int32 visible)
;----------------------------------------------------------------------------
procstartdll    _AF_showCursor

        ARG     dc:DPTR, visible:UINT

        push    ebp
        mov     ebp,esp
        push    ebx

        mov     eax,[visible]
        mov     ebx,[dc]
        call    [AF_devCtx.ShowCursor]

        pop     ebx
        pop     ebp
        ret

procend         _AF_showCursor

;----------------------------------------------------------------------------
; void AF_setMix(AF_devCtx *dc,AF_int32 foreMix,AF_int32 backMix)
;----------------------------------------------------------------------------
procstartdll    _AF_setMix

        ARG     dc:DPTR, foreMix:UINT, backMix:UINT

        push    ebp
        mov     ebp,esp
        push    ebx

        mov     eax,[foreMix]
        mov     ebx,[dc]
        mov     ecx,[backMix]
        call    [AF_devCtx.SetMix]

        pop     ebx
        pop     ebp
        ret

procend         _AF_setMix

;----------------------------------------------------------------------------
; void AF_set8x8Pattern(AF_devCtx *dc,AF_pattern *pattern)
;----------------------------------------------------------------------------
procstartdll    _AF_set8x8MonoPattern

        ARG     dc:DPTR, pattern:DPTR

        push    ebp
        mov     ebp,esp
        push    ebx
        push    esi

        mov     ebx,[dc]
        mov     esi,[pattern]
        call    [AF_devCtx.Set8x8MonoPattern]
```

```
        pop     esi
        pop     ebx
        pop     ebp
        ret

procend         _AF_set8x8MonoPattern

;----------------------------------------------------------------------------
; void AF_setLineStipple(AF_devCtx *dc,AF_stipple stipple)
;----------------------------------------------------------------------------
procstartdll    _AF_setLineStipple

        ARG     dc:DPTR, stipple:S_USHORT

        push    ebp
        mov     ebp,esp
        push    ebx

        mov     ax,[stipple]
        mov     ebx,[dc]
        call    [AF_devCtx.SetLineStipple]

        pop     ebx
        pop     ebp
        ret

procend         _AF_setLineStipple

;----------------------------------------------------------------------------
; void AF_setClipRect(AF_devCtx *dc,AF_int32 minx,AF_int32 miny,
;   AF_int32 maxx,AF_int32 maxy)
;----------------------------------------------------------------------------
procstartdll    _AF_setClipRect

        CallGeneric SetClipRect

procend         _AF_setClipRect

;----------------------------------------------------------------------------
; void AF_drawScan(AF_devCtx *dc,AF_int32 color,AF_int32 y,AF_int32 x1,
;   AF_int32 x2)
;----------------------------------------------------------------------------
procstartdll    _AF_drawScan

        ARG     dc:DPTR, color:UINT, y:UINT, x1:UINT, x2:UINT

        push    ebp
        mov     ebp,esp
        push    ebx
        push    esi

        mov     eax,[y]
        mov     ebx,[dc]
        mov     ecx,[x1]
        mov     edx,[x2]
        mov     esi,[color]
        call    [AF_devCtx.DrawScan]

        pop     esi
        pop     ebx
        pop     ebp
        ret

procend         _AF_drawScan

;----------------------------------------------------------------------------
; void AF_drawPattScan(AF_devCtx *dc,AF_int32 foreColor,AF_int32 backColor,
;   AF_int32 y,AF_int32 x1,AF_int32 x2)
;----------------------------------------------------------------------------
procstartdll    _AF_drawPattScan

        ARG     dc:DPTR, color:UINT, backColor:UINT, y:UINT, x1:UINT, x2:UINT

        push    ebp
        mov     ebp,esp
```

```
        push    ebx
        push    esi
        push    edi

        mov     eax,[y]
        mov     ebx,[dc]
        mov     ecx,[x1]
        mov     edx,[x2]
        mov     esi,[color]
        mov     edi,[backColor]
        call    [AF_devCtx.DrawPattScan]

        pop     edi
        pop     esi
        pop     ebx
        pop     ebp
        ret

procend         _AF_drawPattScan

;----------------------------------------------------------------------------
; void AF_drawScanList(AF_devCtx *dc,AF_color color,AF_int32 y,AF_int32 len,
;   AF_int16 *scans)
;----------------------------------------------------------------------------
procstartdll    _AF_drawScanList

        ARG     dc:DPTR, color:UINT, y:UINT, len:UINT, scans:DPTR

        push    ebp
        mov     ebp,esp
        push    ebx

        mov     eax,[y]
        mov     ebx,[dc]
        mov     ecx,[len]
        mov     esi,[scans]
        mov     edx,[color]
        call    [AF_devCtx.DrawScanList]

        pop     ebx
        pop     ebp
        ret

procend         _AF_drawScanList

;----------------------------------------------------------------------------
; void AF_drawRect(AF_devCtx *dc,AF_color color,AF_int32 left,AF_int32 top,
;   AF_int32 width,AF_int32 height)
;----------------------------------------------------------------------------
procstartdll    _AF_drawRect

        CallGeneric1Color DrawRect

procend         _AF_drawRect

;----------------------------------------------------------------------------
; void AF_drawPattRect(AF_devCtx *dc,AF_color foreColor,AF_color backColor,
;   AF_int32 left,AF_int32 top,AF_int32 width,AF_int32 height)
;----------------------------------------------------------------------------
procstartdll    _AF_drawPattRect

        CallGeneric2Color DrawPattRect

procend         _AF_drawPattRect

;----------------------------------------------------------------------------
; void AF_drawLine(AF_devCtx *dc,AF_color color,AF_fix32 x1,AF_fix32 y1,
;   AF_fix32 x2,AF_fix32 y2)
;----------------------------------------------------------------------------
procstartdll    _AF_drawLine

        CallGeneric1Color DrawLine

procend         _AF_drawLine
```

```
;----------------------------------------------------------------------------
; void AF_drawStippleLine(AF_devCtx *dc,AF_color foreColor,AF_color backColor,
;   AF_fix32 x1,AF_fix32 y1,AF_fix32 x2,AF_fix32 y2)
;----------------------------------------------------------------------------
procstartdll    _AF_drawStippleLine

        CallGeneric2Color DrawStippleLine

procend         _AF_drawStippleLine

;----------------------------------------------------------------------------
; void AF_drawTrap(AF_devCtx *dc,AF_color color,AF_int32 y,AF_int32 count,
;   AF_fix32 x1,AF_fix32 x2,AF_fix32 slope1,AF_fix32 slope2)
;----------------------------------------------------------------------------
procstartdll    _AF_drawTrap

        CallGeneric1Color DrawTrap

procend         _AF_drawTrap

;----------------------------------------------------------------------------
; void AF_drawTri(AF_devCtx *dc,AF_color color,AF_fxpoint *v1,AF_fxpoint *v2,
;   AF_fxpoint *v3,AF_fix32 xOffset,AF_fix32 yOffset)
;----------------------------------------------------------------------------
procstartdll    _AF_drawTri

        CallGeneric1Color DrawTri

procend         _AF_drawTri

;----------------------------------------------------------------------------
; void AF_drawQuad(AF_devCtx *dc,AF_color color,AF_fxpoint *v1,AF_fxpoint *v2,
;   AF_fxpoint *v3,AF_fix32 xOffset,AF_fix32 yOffset)
;----------------------------------------------------------------------------
procstartdll    _AF_drawQuad

        CallGeneric1Color DrawQuad

procend         _AF_drawQuad

;----------------------------------------------------------------------------
; void AF_putMonoImage(AF_devCtx *dc,AF_int32 foreColor,AF_int32 backColor,
;   AF_int32 x,AF_int32 y,AF_int32 byteWidth,AF_int32 height,AF_uint8 *image)
;----------------------------------------------------------------------------
procstartdll    _AF_putMonoImage

        CallGeneric2Color PutMonoImage

procend         _AF_putMonoImage

;----------------------------------------------------------------------------
; void AF_bitBlt(AF_devCtx *dc,AF_int32 left,AF_int32 top,AF_int32 width,
;   AF_int32 height,AF_int32 dstLeft,AF_int32 dstTop,AF_int32 op)
;----------------------------------------------------------------------------
procstartdll    _AF_bitBlt

        CallGeneric BitBlt

procend         _AF_bitBlt

;----------------------------------------------------------------------------
; void AF_bitBltLin(AF_devCtx *dc,AF_int32 srcOfs,
;   AF_int32 left,AF_int32 top,AF_int32 width,AF_int32 height,
;   AF_int32 dstLeft,AF_int32 dstTop,AF_int32 op)
;----------------------------------------------------------------------------
procstartdll    _AF_bitBltLin

        CallGeneric BitBltLin

procend         _AF_bitBltLin

;----------------------------------------------------------------------------
; void AF_srcTransBlt(AF_devCtx *dc,AF_int32 left,AF_int32 top,AF_int32 width,
;   AF_int32 height,AF_int32 dstLeft,AF_int32 dstTop,AF_color transparent)
;----------------------------------------------------------------------------
```

```
procstartdll    _AF_srcTransBlt

        CallGeneric SrcTransBlt

procend         _AF_srcTransBlt

;-----------------------------------------------------------------------------
; void AF_dstTransBlt(AF_devCtx *dc,AF_int32 left,AF_int32 top,AF_int32 width,
;   AF_int32 height,AF_int32 dstLeft,AF_int32 dstTop,AF_color transparent)
;-----------------------------------------------------------------------------
procstartdll    _AF_dstTransBlt

        CallGeneric DstTransBlt

procend         _AF_dstTransBlt

;-----------------------------------------------------------------------------
; void AF_srcTransBltLin(AF_devCtx *dc,AF_int32 srcOfs,
;   AF_int32 left,AF_int32 top,AF_int32 width,AF_int32 height,
;   AF_int32 dstLeft,AF_int32 dstTop,AF_color transparent)
;-----------------------------------------------------------------------------
procstartdll    _AF_srcTransBltLin

        CallGeneric SrcTransBltLin

procend         _AF_srcTransBltLin

;-----------------------------------------------------------------------------
; void AF_dstTransBltLin(AF_devCtx *dc,AF_int32 srcOfs,
;   AF_int32 left,AF_int32 top,AF_int32 width,AF_int32 height,
;   AF_int32 dstLeft,AF_int32 dstTop,AF_color transparent)
;-----------------------------------------------------------------------------
procstartdll    _AF_dstTransBltLin

        CallGeneric DstTransBltLin

procend         _AF_dstTransBltLin

endcodeseg  _vbeaf

endif

        END
```