VESA SVPMI Standard

Video Electronics Standards Association

2150 North First Street, Suite 360
San Jose, CA 95131

Phone: (408) 435-0333
Fax: (408) 435-8225

Super VGA Protected Mode Interface

Standard #VS911020

October 20, 1991

PURPOSE

To standardize a common software interface for protected mode applications (those that do not have access to a real mode BIOS) to VGA video adapters.

SUMMARY

This Standard provides a set of functions which an application program can use to obtain information about the capabilities and characteristics of a specific VGA implementation and to control the operation of such hardware without using a video BIOS.

The goal is to allow an application to take full advantage of the features provided by a variety of display adapters.

Contents

1. Introduction

The IBM VGA[1] is a defacto standard in the PC graphics world. A multitude of different VGA offerings exist in the marketplace, each one providing compatibility with the IBM VGA and implementing various extensions to the VGA standard. These extensions range from higher resolutions and more colors to improved performance and even some graphics processing capabilities.

Several problems face a software developer who intends to take advantage of these "Super VGA"[2] environments. Because there is no standard hardware implementation for the VGA extensions, designing applications for these environments is costly and technically difficult. Except for applications supported by OEM specific display drivers, it is difficult for a software developer to take advantage of the power and capabilities of a variety of Super VGA products.

To remedy this, VESA developed the *VESA VGA BIOS Extension*. Designed for the real mode environment, this standard focuses on returning information about the video environment to the application and assisting the application in initializing and programming the hardware. Unfortunately, due to its design as a set of real mode functions, this standard cannot be used by developers working in a protected mode environment. To address this problem, VESA has defined the *Super VGA Protected Mode Interface*, or SVPMI.

The VESA SVPMI has been developed to minimize the need for board specific device drivers by providing runtime specification of hardware dependent variables and procedures in a text file. The text file will be supplied by the display adapter manufacturer. Applications will be written to load this file during initialization and will use the information contained in the file to determine video capabilities and to set up the graphics hardware in one of the available display modes.

This is expected to be used mainly in protected mode environments such as Unix, OS/2, and Windows, however it may also be used in a real mode environment. The SVPMI is essentially the protected mode "sister" of the VESA Super VGA BIOS Extensions.

Readers of this document should be familiar with programming VGA and Intel [3] Microprocessor hardware.

1.1 Scope of Standard

The SVPMI functions have been designed to put the performance insensitive, nonstandard hardware functions into a text file that is interpreted while putting the performance sensitive,

---

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

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

[3] Intel is a registered trademark of Intel Corporation.

standard hardware functions into the application. This provides portability among VGA systems together with the performance that comes from accessing the hardware directly. In particular, the SVPMI is responsible for mapping video memory into the CPU address space while the application is responsible for performing the actual memory read and write operations.

Functions provided through the SVPMI are limited to manipulating display "modes" and do not include any capabilities such as drawing operations in graphics modes or font handling in text modes. The SVPMI is intended to be used by an application which already has knowledge of the underlying hardware, taking full control of that hardware after using SVPMI functions to establish a particular mode.

The SVPMI has been designed with the idea that an application may assume underlying VGA hardware. It may also be used with other types of hardware, though no such support should be considered a part of the standard.

Though this document refers to the program that interprets SVPMI files as an application, it is expected that the majority of such applications will actually be drivers for environments such as the X Window System, Microsoft Windows, and OS/2 [4].

1.2 Document Overview

This document is intended to be used in two ways. First, by a display adapter manufacturer, as a guide to creating an SVPMI file, and second, by an application developer, as a guide to understanding the format of an SVPMI file to aid in the implementation of an SVPMI interpreter. Sections include introductory material, discussion of the concepts embodied in the standard, and a full description of the contents of an SVPMI file. Also included are several sections to aid in creating SVPMI files and implementing SVPMI interpreters, and a sample SVPMI file.

[4] Microsoft and OS/2 are trademarks of Microsoft Corporation.

2. CPU Video Memory Windows

A standard VGA subsystem provides 256k bytes of memory and a corresponding mechanism to address this memory. Super VGAs and their extended modes require more than the standard 256k bytes of memory but also require that the address space for this memory be restricted to one megabyte due to limitations in the real mode environment. CPU video memory windows provide a means of accessing this extended VGA memory within the standard CPU address space.

The standard VGA CPU address space for 16 color graphics modes is typically at segment 0xA000 for 64k. This gives access to the 256k bytes of a standard VGA, i.e. 64k per plane. Access to the extended video memory is accomplished by mapping portions of the video memory into the standard VGA CPU address space.

Every super VGA hardware implementation provides a mechanism for software to specify the offset from the start of video memory which is to be mapped to the start of the CPU address space. Providing both read and write access to the mapped memory provides the necessary support for an application to manipulate the extended video memory. This section describes how several hardware implementations of CPU video memory windows operate, their impact on application software design, and relates them to the software model presented by the SVPMI.

The SVPMI informs the application of the parameters that control the hardware mechanism of mapping the video memory into the CPU address space and then lets the application control the mapping within those parameters.

2.1 Hardware Window Types

Different hardware implementations of CPU video memory windows can be supported by the SVPMI. The information necessary for an application to understand the type of hardware implementation is provided by the SVPMI to the application. There are three basic types of hardware windowing implementations and they are described below. The types of windowing schemes described do not include differences in granularity or window size.

2.1.1 Single Window Systems

Some hardware implementations only provide a single window. This single window will be readable as well as writeable. However, this may cause a performance degradation when moving data in video memory a distance that is larger than the CPU address space.

Many applications assume that a single read / write 64k window is available. Even if a particular product supports multiple windows, it is advisable to provide a single window mode in the SVPMI file for that product.

2.1.2 Dual Overlapping Windows

Some hardware implementations provide two windows, one for reading and one for writing. In this case, the windows can, and usually do, share the same CPU address space with one window being read only and the other being write only.

2.1.3 Dual Non-Overlapping Windows

Another mechanism used by two window systems is to allow both windows to be read and written. In this case, the windows must be mapped to different CPU addresses.

2.2 Single Window Operation

The organization of most software algorithms which perform video operations consists of a pair of nested loops: an outer loop over rows or scan lines and an inner loop across the row or scan line. The latter is the proverbial inner loop, which is the bottle neck to high performance software.

If a target rectangle is large enough, or poorly located, part of the required memory may be within the video memory mapped into the CPU address space and part of it may not be addressable by the CPU without changing the mapping. It is desirable that the test for remapping the video memory is located outside of the inner loop.

This is typically accomplished by selecting the mapping offset of the start of video memory to the start of the CPU address space so that at least one entire row or scan line can be processed without changing the video memory mapping. There are currently no super VGAs that allow this offset to be specified on a byte boundary and there is a wide range among super VGAs in the ability to position a desired video memory location at the start of the CPU address space.

The number of bytes between the closest two bytes in video memory that can be placed on any single CPU address is defined as the granularity of the window mapping function. Some super VGA systems allow any 4k video memory boundary to be mapped to the start of the CPU address space, while other super VGA systems allow any 64k video memory boundary to be mapped to the start of the CPU address space. These two example systems would have granularities of 4k and 64k, respectively. This concept is very similar to the bytes that can be accessed with a 16 bit pointer in an Intel CPU before a segment register must be changed (the granularity of the segment register or mapping, here is 16 bytes).

It is sometimes required or convenient to move or combine data from two different areas of video memory. If only one window is available, such situations must be separated into phases that operate within one window at a time. For example, a BitBlt operation may copy video data from the lower portion of the display to the upper portion, a range well outside that supported by the window. In this case, the window would first be set to the area to be read, data read into a temporary buffer, then the window would be set to the area to be written and the data written from the temporary buffer to the screen.

2.3 Multiple Window Operation

The above example is much easier to implement if the display adapter being used supports multiple windows. In this case, two windows would be use to map the source and destination simultaneously. Another example of this is storing menus in the video memory beyond the displayed memory because there is hardware support in all VGAs for transferring 32 bits of video data with an 8 bit CPU read and write. Two separately mappable windows must be used in this case if the distance between the source and destination is larger than the size of a single window.

The above example of moving data from one CPU video memory window to another CPU video memory only required read access to one window and only required write access to the other window. Sometimes it is convenient to have read access to both windows and write access to one window. An example of this would be a raster operation where the resulting destination is the source data logically combined with the original destination data.

3. Color Handling

The SVPMI defines a color description mechanism that will allow enough flexibility to fully describe existing and future video adapters' methods for providing color on a display. This color model is based in large part on that used by the X Window System.

In a raster display, individual "dots" on the screen are represented as 1 or more bits of data within a display adapter's video memory. There are many ways to implement the relationship between the data in video memory and how that data is displayed. On the VGA, people commonly think of the "planer" and "packed-pixel" formats, but with the introduction of "HiColor" RAMDAC's which provide no color palette, we can see that a more generalized system to describe color must be used. The SVPMI defines a group of elements to fully describe the relationship between a pixel video memory and what is displayed on a screen.

The first element in the color description is the color model. This describes the basic color handling capabilities of the display adapter such as whether different hues or simply shades of gray are displayed and how the bits that make up a pixel in video memory are used to produce an appropriate color to be displayed on the screen. There are three different color models:

IndexedColor

A IndexedColor model is one in which pixel values are used as indices into a color lookup table. This color lookup table may be modified if the `BitsRGB` value is non-zero. Most VGA display modes follow this color model (when the display adapter is connected to a color display.)

DirectColor

A DirectColor model is one in which pixel values are broken into red, green, and blue components each of which are used as an index into a separate color lookup table. Each of these color lookup tables may be modified if the `BitsRGB` value is non-zero.

GrayScale

A GrayScale model is one in which pixel values are used as indices into a color lookup table that consists of only shades of gray. This lookup table may be modified if the `BitsRGB` value is non-zero. This may be thought of as a IndexedColor model in which the red, green, and blue values for each palette entry are identical. When connected to a analog monochrome monitor, most VGA modes use this color model.

The next element is the BitsPerPixel value. This specifies the total number of bits that are used to represent of one pixel. For example, a standard VGA 4 Plane 16-color graphics mode would have a 4 in this field and a packed pixel 256-color graphics mode would specify 8 in this field.

The next color specification element is NumberOfColors. This specifies the number of colors available to software in the mode. This is not necessarily equal to (2 ^ BitsPerPixel) because some

modes may have bits within a pixel that are unused. For example, a 15 bit color mode would probably have 16 bits per pixel, but only has 32,768 colors.

BitsRGB specifies the number of bits of red, green, and blue are available in the color palette for this display mode. For example, a standard VGA mode would set this value to 6, while a board supporting an 8-bit DAC would set it to 8. If the BitsRGB value is zero, the color lookup table is read-only.

If the color model is DirectColor, the RedSize, RedPosition, GreenSize, GreenPosition, BlueSize, BluePosition, ReservedSize and ReservedPosition elements define the bits that make up the red, green, blue, and reserved component of the pixel value respectively. The bits within each component are contiguous and are not shared with the other components. For example, a 15 bit DirectColor mode where the top bit is unused might have values of (5, 0), (5, 5), (5, 10), (1, 15) for the Red, Green, Blue, and Reserved Size and Position elements respectively. If bit 11 of the `ModeAttributes` value is 1, the memory referenced by the reserved component of the pixel value may be used by the application.

The MemoryModel specifies the general type of memory organization used in this mode such as planar vs. packed pixel.

If the BitsRGB value indicates the color lookup table is read-only, the BlackPixel and WhitePixel elements contain the pixel values that will display black and white on the monitor. While most displays have linearly increasing color ramps so that a pixel of all 0's is black and a pixel of all 1's is white, some display adapters (most notably black and white monochrome displays) use 1 for black and 0 for white, making this definition necessary.

4. SVPMI Files

The SVPMI is based upon a command interpreter which is embedded in an application. The SVPMI interpreter operates in combination with an ASCII data file written in the SVPMI language, which specifies runtime options and parameters for supported graphics display adapters. Parameter settings are grouped by symbolic names in the SVPMI text file.

4.1 File Location

An SVPMI file will be provided on a DOS diskette by the manufacturer of the display adapter supported by the file. This file may be generated by a utility supplied by the manufacturer based on the amount of memory available, the type of DAC, information supplied by the BIOS, etc. As provided on the DOS diskette, the file will be identified with a filename of "\vesa.pmi". The application must provide some mechanism for the user to install the SVPMI file in the protected mode environment.

In the destination environment, the location and name of the SVPMI file is defined by the application.

4.2 File Structure

SVPMI files are made up of a number of sections. There are two types of sections: data and command. Data sections contain statements that provide information to the application in the form of variable assignments. Examples include information describing a particular mode, default palette data if the application does not supply its own colormap, etc. Command sections contain statements to be interpreted by the application. Examples include `AND`, `OUTB`, `ADD`, etc.

The section name specifiers indicate what section of the SVPMI file is to be used by the application, and what data is to be used by the application for a given display mode. The specifiers are used to set up a hierarchical structure in which statements common to a group of sections need only appear at one place in the file.

4.3 Syntax

SVPMI files consist of lines of ASCII text terminated with a carriage return (0xD) and a line feed (0xA). Lines may be no more than 80 characters long.

There is no end of file indication. The DOS end of file character, control-Z (0x1A), is treated as white space. The purpose of these file format conventions is to allow users on both DOS and Unix systems to easily view SVPMI files.

As described above, an SVPMI file is made up of sections. Sections are defined by a token enclosed in square brackets (eg. [TOKEN]) on a line by itself. Within a section, there may be one

or more statements. A statement may appear anywhere on a line in either upper or lower case and are terminated with a semicolon (";"). Statements may not cross lines.

Numeric values may be specified in decimal or hexidecimal. The C language format is used to specify hexidecimal numbers (eg. "0x3D4").

Individual tokens within a statement must be separated by white space, which consists of the following:

a space
a tab character
a comma ,
an open or close parentheses ( )
an open or close brace { }

Comments may appear anywhere in the text. Following C++ language conventions, comments begin with a / / and continue to the end of the current line. Note the distinction between comments in the PMI file and the `[COMMENT]` section which is used to display a comment message to a user.

5. Variables

There are two types of variables used by the SVPMI. The first type is used by the SVPMI interpreter for informational purposes. These variables are set in the SVPMI file's data sections. The second type is used by the interpreter when executing command sections. The SVPMI commands may reference these variables. There are two types of variables used in the command sections:

Variables used in a command are in a form resembling registers, as if the configuration file were a machine with 64 32 bit registers named R0 through R63.

References to absolute locations within the memory buffer for any particular section are made by using the memory pointer variables P0 through Pn, where "n" is the size of the buffer minus 1. Variable P0 refers to the first byte of the buffer, P1 to the next, etc.

6. Video Environment Information

In a real mode environment, the video BIOS is used to provide a common interface to the features provided by a particular board such as extended resolution. Because a protected mode application does not have access to the video BIOS to initialize display modes, it difficult and time consuming to take advantage of any extended features of the underlying hardware.

In most protected mode environments, an application has no standard mechanism to determine what hardware is available. Only by knowing OEM specific features can an application determine the presence of a particular video board. This often involves reading and testing registers located at I/O addresses unique to each OEM. As new boards are introduced, additional code must be added to recognize and support these boards. In many cases, it is not possible to distinguish between boards made by different manufacturers but based on the same VGA chip.

To address this problem, the VESA SVPMI data sections provide information about the video environment. This includes such information as display adapter type, resolution, number of colors, physical memory used, etc.

6.1 Data Sections

Reserved fields have been defined to support future SVPMI extensions and will always be set to zero in this version.

[VERSION]

The version section contains a single number which defines what revision of the SVPMI specification the file refers to. The first revision is 1.0 The SVPMI interpreter built into an application should check the version number to ensure it will be capable of processing the remainder of the file.

This section is required and may only appear once at the beginning of an SVPMI file.

[ACTIVE_ADAPTER]

An SVPMI file may contain information for several adapters. The active adapter section is used to specify which one of the adapters supported in the SVPMI text file is to be used. Other adapter sections in the file are to be ignored. It contains a single line of information defined by the OEM such as the product name of the display adapter to be used.

This section is required and may only appear once in an SVPMI file.

[GRAPHICS_MODE]

The graphics mode specifier is a single number indicating the display mode in which the specified adapter is to be operated if the SVPMI application has no other mechanism to determine which mode to use. Sections containing the commands and data necessary to operate the adapter in this mode must appear elsewhere in the file.

This section is required and may only appear once in an SVPMI file.

[TEXT_MODE]

The text mode specifier is a single number indicating what text mode is to be used on the chosen adapter if the SVPMI application has no other mechanism to determine which mode to use. Sections containing the commands and data necessary to operate the adapter in this mode must appear elsewhere in the file.

This section is required and may only appear once in an SVPMI file.

[ADAPTER]

The adapter section denotes that the following sections, commands and data are specific to the indicated adapter. It contains a single line of information defined by the OEM such as the product name of the display adapter to be used. The statements following this section will only be executed if the adapter named is the one specified in the [ACTIVE_ADAPTER] section.

There may be multiple adapter sections in an SVPMI file.

[COMMENT]

The comment section allows the PMI file to describe any information that is unique to a particular display adapter. The application may choose to display this comment during installation or configuration. In this case, the comment will be displayed in a mono-spaced 7-bit ASCII encoded font. The comment is limited to 15 rows of 40 columns of text each.

The comment section is optional. There may be one comment section for each adapter described in an SVPMI file.

[ADAPTER_INFO]

The adapter info section describes characteristics of the adapter that are common to all modes. The following variables are defined in this section:

BoardType — Indicates what kind of adapter is described by the SVPMI file. Currently defined types are:

EGA	Enhanced Graphics Adapter
VGA	Video Graphics Array

SaveSize — Specifies the number of bytes required to save the state of the adapter. When an application wishes to save the state of the display adapter, it must allocate a buffer of this size and call the function defined by the [SAVESTATE] section.

PaletteSize — Specifies the number of bytes required to save the state of the analog palette. When an application wishes to save the state of the

palette, it must allocate a buffer of this size and either save the palette itself (by assuming underlying VGA hardware) or call the function defined by the [GETPALETTE] section.

There must be an adapter info section for each adapter described in the SVPMI file.

[MODE]

The mode section denotes the beginning of a series of sections that are specific to the indicated display mode. It consists of a single mode number. The [GRAPHICS_MODE] and [TEXT_MODE] sections described earlier in the SVPMI file may be used to select which mode section is to be used, or the application may use other information to make this selection. For example, the application may use an environment variable to allow the user to define the resolution desired. The application would search the modes available in the SVPMI file to find one that matches the resolution specified by the user.

There must be a graphics and text mode section for each adapter described in the SVPMI file.

[MODEINFO]

The modeinfo section describes important characteristics of the mode. The following variables are defined in this section:

ModeAttributes A word of data describing basic information about the mode. Bits within the word have the following meaning:

D0 Reserved
D1 Reserved
D2 Reserved
D3 Monochrome/color mode
0 = Monochrome mode
1 = Color mode
D4 Mode type
0 = Text mode
1 = Graphics mode
D6 Register locking
0 = Registers are not lockable
1 = Registers are lockable
D7 Reserved
D8 Right hand Offscreen memory available.
0 = offscreen memory not available.
1 = offscreen memory available.

This bit is for cases where the BytesPerScanline value exceeds the amount required to contain one scanline of video data. It indicates whether the extra bytes are available for storage and/or panning or not.

D9 Order of bits within a byte.
0 = Least significant bit first.
1 = Most significant bit first.
D10 Order of bytes within a word
0 = Least significant byte first.
1 = Most significant byte first
D11 Reserved bits of DirectColor pixel available.
0 = Memory not available.
1 = Memory available.
D12 Reserved
D13 Reserved
D14 Reserved
D15 Reserved

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

D0 = Window supported
0 = Window is not supported
1 = Window is supported
D1 = Window readable
0 = Window is not readable
1 = Window is readable
D2 = Window writeable
0 = Window is not writeable
1 = Window is writeable
D3-D7 = Reserved

`WinAGranularity` Specifies the smallest boundary, in KB, on which window A can be placed in the video memory.

`WinBGranularity` Specifies the smallest boundary, in KB, on which window B can be placed in the video memory.

`WinASize` Size of window A in KB. A standard VGA 640x480 16 color mode has a value of 64.

`WinBSize` Size of window B in KB

`WinABase` Window A 32 bit base address

`WinBBase` Window B 32 bit base address

`BytesPerScanLine` Specifies how many bytes each logical scanline consists of. The logical scanline could be equal to or larger than the displayed scanline.

`XResolution` Specifies the horizontal resolution in pixels or characters in Graphics and text modes respectively.

`YResolution` Specifies vertical resolution in pixels or characters in Graphics and text modes respectively.

`XCharSize` Specifies character cell width in pixels.

`YCharSize` Specifies character cell height in pixels.

`ColorModel` Specifies the way colors are represented by the display. The following color models are available:

`0`	Gray	Describes a color model in which pixel values are used as indices into a color lookup table that consists of only shades of gray.
`1`	IndexedColor	Describes a color model in which pixel values are used as indices into a color lookup table (RAMDAC).
`2`	DirectColor	Describes a color model in which pixel values are broken into red, green, and blue components each of which are used as an index into a color lookup table..

`BitsPerPixel` Specifies the total number of bits that define the color of one pixel. For example, a standard VGA 4 Plane 16-color graphics mode would have a 4 in this field and a packed pixel 256-color graphics mode would specify 8 in this field.

`NumberOfColors` Specifies the number of colors available to software in that mode. This is not necessarily equal to (2 ^ `BitsPerPixel`) because some modes may have bits within a pixel that are unused. For example, a 15 bit color mode has 16 bits per pixel, but only has 32,768 colors.

`BitsRGB` Specifies the number of bits of red, green, and blue are available in the color palette for this display mode. For example, a standard VGA mode would set this value to 6, while a board supporting an 8-bit DAC would set it to 8. If this value is 0, the color lookup table is read-only.

`RedSize` If the color model is `DirectColor` , this value defines the number of bits that make up the red component of the pixel value.

The bits are contiguous and are not shared with the other components.

`RedPosition` If the color model is `DirectColor` , this value defines the offset from the least significant bit of the pixel value of the red component.

`GreenSize` If the color model is `DirectColor` , this value defines the number of bits that make up the green component of the pixel value. The bits are contiguous and are not shared with the other components.

`GreenPosition` If the color model is `DirectColor` , this value defines the offset from the least significant bit of the pixel value of the green component.

`BlueSize` If the color model is `DirectColor` , this value defines the number of bits that make up the blue component of the pixel value. The bits are contiguous and are not shared with the other components.

`BluePosition` If the color model is `DirectColor` , this value defines the offset from the least significant bit of the pixel value of the blue component.

`ReservedSize` If the color model is `DirectColor` , this value defines the number of bits that make up the reserved component of the pixel value. The bits are contiguous and are not shared with the other components. If bit 11 of the `ModeAttributes` value is 1, the memory referenced by the reserved component of the pixel value may be used by the application.

`ReservedPosition` If the color model is `DirectColor` , this value defines the offset from the least significant bit of the pixel value of the reserved component.

`BlackPixel` If the color lookup table is read-only, the BlackPixel value specifies the pixel value which will produce a "black" color on the display.

`WhitePixel` If the color lookup table is read-only, the WhitePixel value specifies the pixel value which will produce a "white" color on the display.

`NumberOfBanks` The number of banks in which the scanlines are grouped. The remainder from dividing the scanline number by the number of banks is the bank that contains the scanline and the quotient is the

scanline number within the bank. For example, CGA graphics modes have two banks and Hercules graphics mode has four banks. For modes that don't have scanline banks (such as VGA modes 0xD = 0x13), this field should be set to 1.

`BankSize` The size of a bank (group of scanlines) in units of 1 KB. For CGA and Hercules graphics modes, this is 8, as each bank is 8192 bytes in length. For modes that don't have scanline banks (such as VGA modes 0xD - 0x13), this field should be set to 0. Note that in the terminology used here, a "bank" is a group of scanlines while a "window" is a region in video memory.

`MemoryModel` Specifies the general type of memory organization used in this mode. The following models have been defined:

0x0 =	Text mode
0x1 =	CGA graphics
0x2 =	Hercules graphics
0x3 =	VGA 4-plane planar
0x4 =	VGA 8-bit packed pixel
0x5 =	Non-chain 4, 256 color (RIX Mode)
0x6 =	Generic packed pixel
0x7-0xF =	Reserved, defined by VESA
0x10-0xFF =	Reserved, defined by OEM

`ImagePages` Specifies the number of complete display images that will fit into the VGA's memory, at one time, in this mode. The application may load more than one image into the VGA's memory if this field is non-zero, and flip the display between them.

There must be a mode info section for each mode described in the SVPMI file.

`[ADDRESSES]`

The addresses section specifies all physical memory and i/o addresses which need to be accessed in order to operate the display adapter in this mode. Two special types of statements are used in this section:

`MEMORY adr-adr[,adr-adr]`

Declare physical memory locations that this mode needs access to. The declaration is specified as a 32 bit base address, a '"-" delimiter, and a 32 bit ending address. A list of address ranges may be specified by separating the ranges with commas.

`PORT val[-val][,val]`

Declare IO ports that this mode needs access to. A list of values or value ranges may be specified.

For example:

```
MEMORY(0xA0000-0xAFFFF); // 32 bit base address,    32 bit limit
PORT(0x3D4,0x3D5);
PORT(0x3C6-0x3C9);
```

There must be an addresses section for each mode described in the SVPMI file.

[PALETTEDATA]

The palettedata section contains default data used to program the DAC for this mode for applications which do not manage the palette internally. The number of entries in this section is specified by the `NumberOfColors` variable defined in the modeinfo section for this mode. For each color, there are three values: blue, green and red.

For example:

```
0,0,0             // color 0
80,0,0            // color 1
0,80,0            // color 2
80,80,0           // color 3
0,0,80            // color 4
80,0,80           // color 5
0,80,80           // color 6
80,80,80          // color 7

// etc
```

The DAC is always programmed starting with register 0. If the palettedata section is not present, the DAC will not be programmed when the mode is set. It is acceptable for a application to handle palette programming internally, ignoring the information provided in this section.

This section is optional and may appear once per mode.

7. Programming Support

Due to the fact that different Super VGA products have different hardware implementations, application software has great difficulty in adapting to each environment. However, since each is based on the VGA hardware architecture, differences are most often in video mode initialization and memory mapping. The rest of the architecture is usually kept intact, including I/O mapped registers, video buffer location in the CPU address space, DAC location and function, etc.

The SVPMI provides several functions to interface to the different Super VGA hardware implementations. These functions take the form of simple commands that are interpreted by the application. The most important of these sets the display hardware in a particular video mode. This function isolates the application from the details of setting up a video mode on an arbitrary display adapter. Other functions include selecting a bank of video memory to activate, setting palette entries, etc.

7.1 Command Sections

Each of the command sections is actually a function defined by the SVPMI file. When the application wishes to execute one of these functions, it fills in the arguments and calls the SVPMI command interpreter to execute the section. There are at most four arguments and return values passed in the R0 - R3 variables for any SVPMI function. All arguments are assumed to be valid.

On entry to a function, the variables R0 - R63 are uninitialized except in cases where a variable is being used to pass an argument to the function. For functions that reference the pointer variables P0 - Pn, the variable P0 will point to the first byte of the buffer, P1 to the next, etc.

[MODEDETECT]

The mode detect section is a set of commands which can be executed to determine if the specified mode can be run on the current hardware configuration. The application should always execute the mode detect section before using any other command sections for a particular mode.

Input:

(none)

Output:

R0 = 1 If mode is available

There must be a mode detect for each mode defined in the SVPMI file. It is acceptable for the mode detect section to simply assign the variable R0 to one if it is difficult or unnecessary to check for the availability of a given mode.

Where possible, it is desirable that the mode detect section not change the display seen by

the user in any way. The mode detect section must always leave the display in the same state as it was in at the beginning of the function.

[SETMODE]

The set mode section contains the commands necessary to set the display adapter into the mode specified in the nearest preceding [MODE] section. This section should set all display adapter registers required for the mode with the exception of the analog palette.

Input:

(none)

Output:

(none)

There must be a set mode section for each mode defined in the SVPMI file.

[GETCOLOR]

The get color section contains the commands necessary to retrieve one color from the analog palette.

Input:

R0 = Palette index of color desired.

Output:

R0 = Red component of color.
R1 = Green component of color.
R2 = Blue component of color.

The get color section should appear once for each mode that has a different color access mechanism from the VGA standard.

[SETCOLOR]

The set color section contains the commands necessary to set one color in the analog palette.

Input:

R0 = Palette index of color to be set.
R1 = Red component of color.
R2 = Green component of color.
R3 = Blue component of color.

Output:

(none)

The set color section should appear once for each mode that has a different color access mechanism from the VGA standard.

[GETPALETTE]

The get palette section contains the commands necessary to save the analog palette. This will be used mainly when saving the entire state of the display adapter.

Input:

P0 = Pointer to a buffer large enough to save the entire palette as defined by the `PaletteSize` variable in the `[ADAPTER_INFO]` section.

Output:

(none)

The get palette section should appear once for each mode that has a different color access mechanism from the VGA standard.

[SETPALETTE]

The set palette section contains the commands necessary to restore the analog palette.

Input:

P0 = Pointer to a buffer containing the palette data.

Output:

(none)

The set palette section should appear once for each mode that has a different color access mechanism from the VGA standard.

[GETWINDOW]

The get window section contains the commands necessary to find the current position of the specified window in video memory. Only simple commands may be included in the getwindow section.

Input:

R0 = 0 to retrieve the position of Window A
1 to retrieve the position of Window B

Output:

R0 = Position of window `WinGranularity` units

There must be a get window section for each mode described in the SVPMI file that supports windows unless it is impossible to determine the current window from the hardware.

[SETWINDOW]

The set window section contains the commands and data necessary to set the position of the specified window in video memory. Only simple commands may be included in the set window section. For performance reasons, it is important that this section do as little as is absolutely necessary to set the window to the specified location because it will be

called during actual drawing operations by the application. In systems supporting two windows, only the position of the window specified may be modified. The other window must remain unchanged.

Input:

R0 = Position of window `WinGranularity` units
R1 = 0 to set the position of Window A
1 to set the position of Window B

Output:

(none)

There must be a set window section for each mode described in the SVPMI file that supports windows.

[SAVESTATE]

The savestate section contains the commands necessary to completely save the current state of the VGA's registers with the exception of the analog palette registers. All of the commands in this section will use variables P0 through Pn as virtual pointers to the save buffer.

Input:

P0-Pn= Pointer to buffer large enough to save the entire state based on the size specified in the `SaveSize` variable of the `[ADAPTER_INFO]` .

Output:

(none)

For example:

```
//
// #typedef savebuf {
//         char crtc_index;
//         char crtc_vals[0x18];
//         etc.
// };
//

inb(R0, 0x3D4);                          // save the CRTC index
P0 = R0;
bufindexinb(P1, 0x3D4, 0x3D5, 0x18);     // save the CRTC registers
```

[RESTORESTATE]

The restorestate section contains the commands necessary to completely restore the state of the VGA's registers from the save buffer with the exception of the analog palette registers.. All of the commands in this section will use variables P0 through Pn as virtual pointers to the save buffer.

Input:

P0-Pn= Pointer to buffer containing previously saved state.

Output:
(none)

For example:

```
bufindexoutb(P1, 0x3D4, 0x3D5, 0x18);    // restore CRTC registers
R0 = P0;
out(0x3D4, R0);                          // restore CRTC index
```

[UNLOCKREGS]

The unlockregs section contains the commands necessary to allow access to all of the VGA's registers.

Input:
(none)

Output:
(none)

This section will only be present if Bit D6 of the `ModeAttributes` field in the `[MODEINFO]` section is set.

[LOCKREGS]

The lockregs section contains the commands necessary to prevent access to some or all of the VGA's registers.

Input:
(none)

Output:
(none)

This section will only be present if Bit D6 of the `ModeAttributes` field in the `[MODEINFO]` section is set.

8. Commands

Unless otherwise noted, arguments specified with commands may be either variables or numeric values.

8.1 Simple Commands

Simple commands may appear in any command section. With the exception of the assignment command, the memory pointer variables (P0 - Pn) may not be used in simple commands.

`var = val`
Assign a value to a variable.

`AND dst, bits`
Bitwise AND of variable specified by dst with bits. Result placed in dst.

`OR dst, bits`
Bitwise OR of variable specified by dst with bits. Result placed in dst.

`NOT dst`
Bitwise NOT of variable specified by dst.

`XOR dst`
Bitwise XOR of variable specified by dst.

`SHR dst, amount`
Logical shift right of variable specified by dst by amount. Result placed in dst.

`SHL dst, amount`
Logical shift left of variable specified by dst by amount. Result placed in dst.

`ADD dst, val`
Add a value to a variable. Result placed in dst.

`SUB dst, val`
Subtract a value from a variable. Result placed in dst.

`MUL dst, val`
Multiple a variable by a value. Result placed in dst.

`DIV dst, val`
Integer divide a variable by a value. Result placed in dst.

`REM dst, val`
Divide a variable by a value, place remainder in dst.

READB dst, src
Read a byte from 32 bit address in src into variable specified by dst.

READW dst, src
Read a word from 32 bit address in src into variable specified by dst.

READDW dst, src
Read a double word from 32 bit address in src into variable specified by dst.

WRITEB dst, src
Write a byte from variable specified by src to 32 bit address in dst.

WRITEW dst, src
Write a word from variable specified by src to 32 bit address in dst.

WRITEDW dst, src
Write a double word from variable specified by src to 32 bit address in dst.

OUTB port, data
Output a byte to an IO port.

OUTW port, data
Output a word to an IO port.

OUTDW port, data
Output a doubleword to an IO port.

INB dst, port
Input a byte from an IO port to variable specified by dst.

INW dst, port
Input a word from an IO port to variable specified by dst.

INDW dst, port
Input a double word from an IO port to variable specified by dst.

8.2 Complex Commands

Complex commands are limited to the following sections: [GETPALETTE] , [SETPALETTE] , [SAVESTATE] , [RESTORESTATE] , [SETMODE] .

BOUTB count, index port, data port
Block byte output of R0 to R(count - 1). This command first outputs the register number to the index port, then outputs the contents of the register to the data port. The register number is incremented, and the operation is repeated.

BUFOUTB src, port, count

Block byte output from data buffer. This command outputs count number of bytes to the data port. The data is read beginning at the memory pointer `src`.

BUFINB dst, port, count

Block byte input to data buffer. This command inputs count number of bytes from the data port. The data is written beginning at the memory pointer `dst`.

BUFINDEXOUTB src, index port, data port, count

Block byte output from data buffer. This command first outputs an index (beginning with 0) to the index port, then outputs the data pointed to by `src` to the data port. The index port and `src` pointer are incremented, and the operation is repeated `count` times.

BUFINDEXINB dst, index port, data port, count

Block byte input to data buffer. This command first outputs an index (beginning with 0) to the index port, then inputs a byte from the data port, storing the result in the buffer pointed to by `dst`. The index port and `dst` pointer are incremented, and the operation is repeated `count` times.

9. SVPMI File Structure

A SVPMI file contains four types of objects:

Section Name Specifiers
Commands
Data
Comments

The following sequence illustrates the basic organization of a SVPMI text file:

```
//
// These first three sections occur only once in the file.
///

[ACTIVE_ADAPTER]
          acme deluxe

[GRAPHICS_MODE]
          102                 // 800x600-16

[TEXT_MODE]
          3                   // 80 x 25

//
// The following sections repeat for various adapters and modes.
///

[ADAPTER]
          acme deluxe

[COMMENT]
Be sure the switches are set correctly
for the type of monitor you're using.

[MODE]
          102

[SectionName1]                // for this section
command1  parameter           // do these commands
command2  parameter
                              // etc.

[SectionName2]
data,data,data.....
                              // etc.

[SectionName3]
command1  parameter
command2  parameter
                              // etc.


[MODE]
```

```
          103                 // 800x600-256

[SectionName1]
command1  parameter
command2  parameter
                              // etc.

[SectionName2]
data,data,data...
                              // etc.

[SectionName3]
command1  parameter
command2  parameter
                              // etc.
[ADAPTER]
          acme super-deluxe
[MODE]
          102
[SectionName1]
                              // etc.
```

10. Creating SVPMI Files

This section is targeted at the person who must create a new SVPMI file.

Because an SVPMI file supporting all possible modes for a particular display adapter would be extremely large, it is recommended that a program be written that will generate the appropriate SVPMI file based on the modes a user wishes to have available. This program might generate the desired file based on pieces chosen from a "master" SVPMI file, or perhaps could create the file by dumping the registers following a BIOS mode set.

For modes which require video memory windows, a single 64k read / write window should be the default. Other window schemes may also be provided, but the single 64k window should be provided if possible.

Some 16 color drivers require linear addressing within a plane of memory. For this reason, resolutions requiring more than 64k of memory for a single plane should provide a 128k plane mode if the hardware supports such an option.

It is important to make the set window function as small as possible. The efficiency of this function can have a dramatic effect on the performance of some rendering functions.

11. Implementing an SVPMI Interpreter

This section is targeted to the application developer who will need to implement an interpreter to make use of SVPMI files.

There are many ways to approach the development of an SVPMI interpreter. The two most obvious approaches are converting an SVPMI file to an internal form ("p-code") and compiling the information contained in the command sections of an SVPMI file into actual machine language for the environment in use.

Each of these methods has additional variations. For instance, an application may choose to load the SVPMI file at initialization time, search for the sections of interest, and either convert or compile only those sections for use later in the execution of the application. Another approach would be to use a separate program to convert or compile the entire SVPMI file, storing the results in a separate file. The actual application would then use this newly created file to perform SVPMI-related functions.

12. Sample VGA SVPMI File

```
//
// vga.pmi - SVPMI File for standard VGA adaptor
//
//        640x480-16          graphics
//        80x25               text
//
//
// Video Electronics Standards Association  (File author's company name here.)
// 1330 South Bascom Avenue, Suite D
// San Jose, CA  95128-4502
//

[VERSION]
          1.0;

[ACTIVE_ADAPTER]
          VGA;

[GRAPHICS_MODE]
          0x12;

[TEXT_MODE]
          3;

[ADAPTER]
          VGA;

[ADAPTER_INFO]
          BoardType = VGA;
          SaveSize  = 61;
          PaletteSize         = 768;

//
// 640x480x4
//
[MODE]
          0x12;

[MODEINFO]
          ModeAttributes      = 0xA18;
          WinAAttributes      = 0;
          WinBAttributes      = 0;
          WinAGranularity     = 0;
          WinASize            = 64;
          WinABase            = 0xa0000;
          BytesPerScanLine    = 80;
          XResolution         = 640;
          YResolution         = 480;
          XCharSize           = 0;
          YCharSize           = 0;
          ColorModel          = 1;
          BitsPerPixel        = 4;
          NumberOfColors      = 16;
```

```
          BitsRGB             = 6;
          NumberOfBanks       = 1;
          BankSize            = 0;
          MemoryModel         = 0x3;
          NumberOfImagePages  = 1;

[ADDRESSES]
          MEMORY(0xA0000 - 0xAFFFF);
          PORT(0x3D4, 0x3D5);                     // for CRTC regs
          PORT(0x3C4, 0x3C5);                     // for SEQ regs
          PORT(0x3c2);                            // Misc Output reg
          PORT(0x3DA);                            // Addribute Controller
          PORT(0x3C0);                            // Palette enable
          PORT(0x3CE, 0x3CF);                     // graphics controller regs

[MODEDETECT]
          r0 = 1;

[SETMODE]
          inb(r63, 0x3da);                        // reset attr F/F
          outb(0x3c0, 0);                         // disable palette
          outb(0x3d4, 0x11); outb(0x3d5, 0);      // unprotect crtc regs 0-7

          //
          // Reset and set sequencer registers
          //
          r0 = 0x01;  r1 = 0x01;  r2 = 0x0f;  r3 = 0x00;
          r4 = 0x06;
          boutb(5, 0x3c4, 0x3c5);                 // reset, sequencer regs

          //
          // Set misc out register
          //
          outb(0x3c2, 0xef);

          r0=3;
          boutb(1, 0x3c4, 0x3c5);                 // sequencer enable

          //
          // Set all crtc registers
          //
          r0  = 0x5f;  r1  = 0x4f;  r2  = 0x50;  r3  = 0x82;
          r4  = 0x54;  r5  = 0x80;   r6  = 0x0b;  r7  = 0x3e;
          r8  = 0x00;  r9  = 0x40;  r10 = 0x00;  r11 = 0x00;
          r12 = 0x00;  r13 = 0x00;  r14 = 0x00;  r15 = 0x00;
          r16 = 0xea;  r17 = 0x0c;  r18 = 0xdf;  r19 = 0x28;
          r20 = 0x00;  r21 = 0xe7;  r22 = 0x04;  r23 = 0xe3;
          r24 = 0xff;
          boutb(25, 0x3d4, 0x3d5);

          //
          // Set all graphics controller registers
          //
          outb(0x3cc, 0); outb(0x3ca, 1);
          r0 = 0x00;  r1 = 0x00;  r2 = 0x00;  r3 = 0x00;
          r4 = 0x00;  r5 = 0x00;  r6 = 0x05;  r7 = 0x0f;
          r8 = 0xff;
```

```
          boutb(9, 0x3ce, 0x3cf);

          //
          // Set all  attribute registers
          //
          inb(r63, 0x3da);                          // reset flip/flop
          r0  = 0x00;  r1  = 0x01;  r2  = 0x02;  r3  = 0x03;
          r4  = 0x04;  r5  = 0x05;  r6  = 0x14;  r7  = 0x07;
          r8  = 0x38;  r9  = 0x39;  r10 = 0x3a;  r11 = 0x3b;
          r12 = 0x3c;  r13 = 0x3d;  r14 = 0x3e;  r15 = 0x3f;
          r16 = 0x01;  r17 = 0x00;  r18 = 0x0f;  r19 = 0x00;
          r20 = 0x00;
          boutb(21, 0x3c0, 0x3c0);

          outb(0x3c0, 0x20);                        // enable palette


//
// Text Mode
//
[MODE]
          3;

[MODEINFO]
          ModeAttributes      = 0x8;
          WinAAttributes      = 0;
          WinAGranularity     = 0;
          WinASize            = 0;
          WinABase            = 0;
          BytesPerScanLine    = 80;
          XResolution         = 80;
          YResolution         = 25;
          XCharSize           = 9;
          YCharSize           = 16;
          ColorModel          = 3;
          BitsPerPixel        = 4;
          NumberOfColors      = 16;
          BitsRGB             = 6;
          NumberOfBanks       = 1;
          BankSize            = 0;
          MemoryModel         = 0;
          NumberOfImagePages  = 1;

[ADDRESSES]
          MEMORY(0xA0000 - 0xAFFFF);
          PORT(0x3D4, 0x3D5);                       // for CRTC regs
          PORT(0x3C4, 0x3C5);                       // for SEQ regs
          PORT(0x3c2);                              // Misc Output reg
          PORT(0x3DA);                              // Addribute Controller
          PORT(0x3C0);                              // Palet te enable
          PORT(0x3CE, 0x3CF);                       // graphics controller regs

[MODEDETECT]
          r0 = 1;

[SETMODE]
          inb(r63, 0x3da);                          // reset attr F/F
```

```
outb(0x3c0, 0);                          // disable palette
outb(0x3d4, 0x11); outb(0x3d5, 00);      // unprotect crtc regs 0-7

//
// Reset and set sequencer registers
//
r0 = 0x01;  r1 = 0x00;  r2 = 0x03;  r3 = 0x00;// sequencer regs
r4 = 0x02;
boutb(5, 0x3c4, 0x3c5);

//
// Set misc out register
//
outb(0x3c2, 0x63);                       // misc out reg

r0 = 0x03;
boutb(1, 0x3c4, 0x3c5);                  // sequencer enable

//
// Set all crtc registers
//
r0  = 0x5f;  r1  = 0x4f;  r2  = 0x50;  r3  = 0x82;
r4  = 0x55;  r5  = 0x81;  r6  = 0xbf;  r7  = 0x1f;
r8  = 0x00;  r9  = 0xc7;  r10 = 0x0c;  r11 = 0x0e;
r12 = 0x00;  r13 = 0x00;  r14 = 0x07;  r15 = 0x80;
r16 = 0x9c;  r17 = 0x8e;  r18 = 0x8f;  r19 = 0x28;
r20 = 0x1f;  r21 = 0x96;  r22 = 0xb9;  r23 = 0xa3;
r24 = 0xff;
boutb(25, 0x3d4, 0x3d5);

//
// Set all graphics controller registers
//
outb(0x3cC, 00); outb(0x3ca, 01);
r0 = 0x00;  r1 =  0x00;  r2 = 0x00;  r3 = 0x00;
r4 = 0x00;  r5 = 0x10;  r6 = 0x0e;  r7 = 0x00;
r8 = 0xff;
boutb(9, 0x3ce, 0x3cf);

//
// Set all attribute registers
//
inb(r63, 0x3da);                          // reset flip/flop
r0  = 0x00;  r1  = 0x01;  r2  = 0x02;  r3  = 0x03;
r4  = 0x04;  r5  = 0x05;  r6  = 0x14;  r7  = 0x07;
r8  = 0x38;  r9  = 0x39;  r10 = 0x3a;  r11 = 0x3b;
r12 = 0x3c;  r13 = 0x3d;  r14 = 0x3e;  r15 = 0x3f;
r16 = 0x0c;  r17 = 0x00;  r18 = 0x0f;  r19 = 0x08;
r20 = 0x00;
boutb(21, 0x3c0, 0x3c0);

outb(0x3c0,  0x20);                      // enable palette
```