SciTech SuperVGA Kit

Cross Platform Development Tools for DOS, Windows 3.1 and Windows '95

Copyright © 1996 SciTech Software.
All Rights Reserved.

Version 6.0
May 1996

Contents

Introduction

This document provides both an overview and reference manual for the SciTech SuperVGA Kit and associated libraries. Note that the SuperVGA Kit distribution includes an entire suite of tools for developing cross platform, fullscreen SuperVGA graphics applications under DOS, Windows 3.1 and Windows '95.

Included in this suite is the SuperVGA Kit itself, the WinDirect libraries for fullscreen SuperVGA graphics under Windows, the AVIDirect libraries for fullscreen AVI file playback, the PM/Pro library for providing a DOS extender independant API for protected mode services and the Zen Timer for providing high precision timing under all these environments.

Is this product for you?

The primary purpose of this SDK is to provide software developers with a comprehensive source of programming information for developing low-level, high performance, cross platform SuperVGA graphics software. It is not intended to provide the functionality of a complete graphics library, but to provide the basic groundwork functions necessary to get an application up and running properly on VBE compliant devices and the necessary framework for writing directly to a linear framebuffer.

If you require a more complete graphics library that supports complex graphical primitives, then this library is not for you. In this case a full featured graphics library such as the SciTech MegaGraph Graphics Library (MGL) would probably be more suitable. For more information on the MGL please consult the product release documentation that was provided with this library.

Installation

Hardware and Software Requirements

The SuperVGA Kit requires the following minimum runtime requirements for programs that you will be developing:

- IBM PC compatible
- An 80386 or higher processor
- VGA or SuperVGA display adapter
- MS-DOS, Windows 3.1 or Windows '95

Installing the SciTech SuperVGA Kit

Before you install any SciTech Software Products, you should decide upon a standard root directory for installing all of the products into. By default the installation programs will choose the C:\SCITECH directory and the installation location. You might like to install the files onto a different drive, but should install all the files for all the different distributions (SuperVGA Kit, MGL for Windows, freeware utilities etc) that you have under the same directory tree. Many SciTech Software products use common libraries and common header files (like the PM/Pro library) to get things done. When you install them into the same directory you will only have one copy of each of these common files and won't run into conflicts with multiple copies of the same files on your system.

The SuperVGA Kit is supplied as a Windows Self Installing Executeable program. To install the SuperVGA Kit start your copy of Windows and choose Run from the File Menu of the Program Manager (the Start Menu for Windows '95 users). If you have downloaded the file from the internet or from an electronic distribution site the name of the installation executable will be the name of the file you downloaded onto your computer (probably SVGAKTxx.EXE where xx is the version number of the package). If you obtained the SuperVGA Kit on floppy disk, the installation program will be named INSTALL.EXE on the first distribution disk.

Select the installation archive located on your computer or on the floppy disk and run it. Follow the installation instructions on the screen and select the SuperVGA Kit components that you wish to install.

Setting Up Your Compiler Configuration

Once you have installed the files you want from the distribution, you will need inform your compiler where the include files and library files are located. The following steps provide a guide to how to set things up correctly for your compiler:

The installation program would have installed all include files into the d:\scitech\INCLUDE directory on your hard disk, where 'd:\scitech' is the default installation directory that you chose to install the product into. If you are compiling your applications from the Integrated Development Environment (IDE) for your compiler, you will need to set the include directories for your project file's to include the d:\scitech\INCLUDE directory.

If you are compiling from the command line, you simply need to add the d:\scitech\INCLUDE path to your INCLUDE path environment variable (for Borland C++ users you will need to add this directory to your Borland C++ 'turboc.cfg' and 'bcc32.cfg' configuration files locateed in the x:\bc\BIN directory where 'x:\bc' is where you installed the compiler into).

The installation program would have installed all the libraries files for the compilers that you selected under the d:\scitech\LIB directory, where 'd:\scitech' is the default installation directory that you chose to install the product into. Beneath this directory is a hierarchy of directories containing library files for different operating systems and different compilers as follows (some may not be present depending on what libraries your installed and what libraries are supported by your product):

16 bit DOS real mode support:	
DOS16\BC	Borland C++ 4.52 16 bit DOS libraries
DOS16\BC3	Borland C++ 3.1 16 bit DOS libraries
DOS16\SC	Symantec C++ 7.1 16 bit DOS libraries
DOS16\VC	Microsoft Visual C++ 1.52 16 bit DOS libraries
DOS16\WC	Watcom C++ 10.5 16 bit DOS libraries
32 bit DOS protected mode support:	
DOS32\BC	Borland C++ 4.52 32 bit DOS libraries
DOS32\WC	Watcom C++ 10.5 32 bit DOS libraries
16 bit Windows support:	
WIN16\BC	Borland C++ 4.52 Win16 libraries
WIN16\SC	Symantec C++ 7.1 Win16 libraries
WIN16\VC	Microsoft Visual C++ 1.52 Win16 libraries
WIN16\WC	Watcom C++ 10.5 Win16 libraries
32bit Windows support:	
WIN32\BC	Borland C++ 4.52 Win32 libraries
WIN32\SC	Symantec C++ 7.1 Win32 libraries
WIN32\VC	Microsoft Visual C++ 4.0 Win32 libraries
WIN32\WC	Watcom C++ 10.5 Win32 libraries

Note that the compiler versions listed are those that were used to compile the library files that you will find in those directories. In most cases the libraries should work fine for previous versions of the compiler for the standard C libraries (for C++ libraries such as the Techniques Class Library and MGL Plus Pack libraries you may need to recompile them with your compiler).

If you are compiling your applications from the IDE for your compiler, you will need to set the library directories for your project file to include the d:\scitech\LIB\xxx\xx directory (select the appropriate directory from the list above).

If you are compiling from the command line, you simply need to add the d:\scitech\LIB\xxx\xxx path to your LIB path environment variable (for Borland C++ users you will need to add this directory to your Borland C++ 'tlink.cfg' and 'tlink32.cfg' configuration files).

NOTE Watcom C++ Users: By default Watcom C++ compiles all source code using register based parameter passing, so by default all SciTech Software libraries are compiled with register based parameter passing. If you are compiling and linking you code for stack based parameter passing, you will need to link with a different set of libraries. All libraries are provided with both stack and register based versions for Watcom C++ and the default libraries use register based parameter passing. The stack based libraries will have the same name as the register based versions of the libraries, but will have an extra 's' appended to the front of the library name. Hence the SuperVGA Kit library for stack based parameters is called SSVGA.LIB rather than SVGA.LIB.

Once you have done this, you can simply start using the library files as provided. If you intend to re-compile any of the sample programs using the supplied makefiles from the command line, you will need to follow the additional steps outlined below.

Using the Optional SciTech Software Makefile Utilities

In order to be able to re-compile any of the sample programs using the supplied makefiles from the command line, or re-build any of the libraries that come with source code, you will also need to install the SciTech Software Makefile Utilities package (re-run the installation program and install this if you have not done so already). This installs all of the relevant executable utility files (including the freeware make program called DMAKE), batch files and DMAKE startup files required to re-compiled the examples for any of the supported compilers.

The makefile utilities package was developed by SciTech Software to allow us to build all of our code from the command line for any of the supported compilers and operating systems using a common set of makefiles. This is achieved by using a standard make program that supports powerful make startup scripts which are changed to reflect the currently selected compiler. On top of that we provide a number of utility programs and batch files that can be used to fully automate this process. Although the steps involved in getting the Makefile Utilties up and running may seem complicated, it is definately worth it in the long run.

Once you have installed the makefile utilities onto your hard drive, you will need to perform the following steps to set things up properly:

Ensure that you have enough environment space to include all the environment variables that you will need. In practice we find you need around 2048 bytes of environment space, but you may get by with less. To change this with a normal DOS configuration, add the /E:2048 command line switch to the end of your SHELL= command line. If you are using the latest 4DOS from JP Software (highly recommended) then you can do this in the 4DOS startup files.

Change the default executable file path in your AUTOEXEC.BAT file to include the d:\scitech\BIN directory (where 'd:\scitech' is where you installed the MGL files into). This can be placed anywhere on your path, so long as the DMAKE.EXE file in the BIN directory will be found first (if there is another program with the same name).

Add the environment variable SCITECH to your AUTOEXEC.BAT file. The SCITECH environment variable is used by the batch files in the BIN directory for setting up for compiling with a particular compiler, and by the DMAKE program so that it can find all of the relevant files during compilation (such as include files and libraries files).

Edit the d:\scitech\BIN\DOS-VARS.BAT batch file to set up the enivornment variables needed by the remainder of the utility batch files. This file is an example that we use for DOS, so you can start with this to build your own configuration file. This should be the only batch file in the BIN directory that should need to be modified as all the remaining batch files feed off the environment variables set up by this master batch file. This batch file essentially lets the rest of the system know where you have installed all of your compiler specific files such as your include and library files, and where your CD-ROM is located (so you can have some files offline on the CD-ROM).

You should now add this batch file to your AUTOEXEC.BAT file if you will use it all the time, or run this batch file before you use any of the remaining utilities.

Run the relevant batch file to set up the environment for your compiler according to the list below. These files require the SCITECH environment variable to be setup correctly, and the relevant variables from the previous step to be set up correctly. You should not need to edit the batch files themselves, but you *will* need to edit the DOS-VARS.BAT file to set things up correctly before these batch files can be run. A number of batch files are supplied to support the different compiler configurations as follows (some libraries may not support a particular compiler depending on language requirements):

16 bit DOS real mode support:	
bc3-dos.bat	Borland C++ 3.1 16 bit
bc16-dos.bat	Borland C++ 4.x 16 bit
vc16-dos.bat	Microsoft Visual C++ 1.x 16 bit
sc16-dos.bat	Symantec C++ 6.x/7.x 16 bit
wc16-dos.bat	Watcom C++ 10.x 16 bit
32 bit DOS protected mode support:	
bc32-dos.bat	Borland C++ 4.x 32 bit DPMI32
bc32-tnt.bat	Borland C++ 4.x 32 bit PharLap TNT
wc32-dos.bat	Watcom C++ 10.x 32 bit DOS4GW/PMODEW
wc32-tnt.bat	Watcom C++ 10.x 32 bit PharLap TNT
wc32-x32.bat	Watcom C++ 10.x 32 bit FlashTek X32/X32-VM
16 bit Windows support:	
bc16-win.bat	Borland C++ 4.x 16 bit
vc16-win.bat	Microsoft Visual C++ 1.x 16 bit

sc16-win.bat	Symantec C++ 6.x/7.x 16 bit
wc16-win.bat	Watcom C++ 10.x 16 bit
32 bit Windows support:	
bc32-win.bat	Borland C++ 4.x 32 bit
vc32-win.bat	Microsoft Visual C++ 1.x/2.x 32 bit
sc32-win.bat	Symantec C++ 6.x/7.x 32 bit
wc32-win.bat	Watcom C++ 10.x 32 bit

NOTE Borland C++ Users: When using Borland C++ 3.1/4.x with these batch files, you will need to edit the supplied TURBOC.*, TLINK.*, BCC32.* and TLINK32.* to contain the proper information for your installation. These files are then copied by the batch files into the proper Borland C++ installation directories to correctly set up the compiler for compiling and linking code for the specified target environment.

Make sure that you edit these files correctly *before* you run the supplied batch files for the first time, otherwise you will clobber your previous configuration files.

One you have everything set up, you should be able to run DMAKE from the directory containing the sample programs that you wish to compile. If things run smoothly you should get a resulting executable file that you can run.

Standard Makefile Targets

All of the makefile utilities DMAKE startup scripts support a standard set of make targets for controlling the compilation for the current makefile. The following is a list of the most common and useful ones that you will need to use when building examples and re-compiling any libraries:

dmake	Running dmake by itself in a directory will select the default target for the makefile, which is usually to compile and link all sample programs in that directory. Some makefiles only support building libraries so the default target may produce an error.
dmake -u	The -u command line options forces a complete re-build of all files so is useful to re-build and entire directory from scratch.
dmake lib	This re-builds the library for the directory.
dmake install	This re-builds the library for the directory and installed the final library into the appropriate d:\scitech\LIB\xxx\xx directory. You should only do this once you are *sure* that everything is working correctly! The old library will simply be overwitten by the new library.
dmake clean	This cleans out all object files, librraies, pre-compiled header files etc from the directory, but leaves all executable files and DLL’s.
dmake cleanexe	This cleans out all non-source files including all DLL’s and EXE files.

Standard Makefile Options

All of the makefile utilities DMAKE startup scripts support a standard set of make options for controlling the way that the compilation is performed. Makefile options are provided for turning on debug information, speed or size optimizations and inline floating point instructions. By default when you build files, no optimisations and no debugging information is generated. The following is a list of the most common and useful ones that you will need to use when building examples and re-compiling any libraries:

DBG	Turns on debugging information.
OPT	Turns on speed optimisations.
OPT_SIZE	Turns on size optimisations.
FPU	Turns on inline floating point arithmetic
STKCALL	Turns on stack calling conventions for Watcom C++

To pass the options to DMAKE, you can do it in one of two ways. You can either pass the options on the command line or you can set the options as global environment variables and then run DMAKE. For instance the following are equivalent:

```
dmake DBG=1 OPT=1 install
```

or

```
set DBG=1
set OPT=1
dmake install
```

Assembling 32 bit code

All of SciTech Software's assembler code is written in Borland TASM IDEAL mode, so you will need a copy of Borland TASM in order to re-assemble the assembler code. If you are assembling for 32 bit protected mode, you *MUST* use TASM 4.0 or later, since TASM 3.1 and earlier do not generate correct 32 bit code in some instances. If you dont have a copy of Borland TASM but you wish to re-build the C code portions of the libraries, you can recompile and link with just the module you need, or you can use your compilers librarian utility to extract the pre-assembled modules from the libraries that you wish to build (see your compilers documentation for more information).

Changing the default DOS Extender

All of the SciTech Software DOS libraries are DOS Extender independent. All DOS extender dependant information is encapsulated in the PMODE.LIB library files. The default library provided for each of the compilers is compiled for the default DOS extender normally used by that compiler. All you need to do in order to use a different DOS extender is re-compile the PM/Pro library with the appropriate command line options, and then link with this new library.

SciTech Standard Directory Tree

All SciTech Software products install into a common directory structure so that all header files and library files are all stored in common locations. This makes it very easy to find particular library files and include files that you need, but it also means that once you have set yourself up for compiling and linking with one SciTech Software product, installing and using another is simple because everything will already be set up.

The following is a brief outline of the common directory tree structure:

```
SCITECH              ROOT directory (SCITECH is default official name)
│
├────BIN             Useful tools and batch files for library building
│
├────DOC             Documentation files. Each individual product has its
│    ├────MGL        own separate directory which contains all files
│    ├────...        specific to that product.
│    └────SVGAKIT
│
├────REDIST          All redistributable DLL and EXE components
│
├────FONTS           All MGL font files installed here.
│
├────BITMAPS         All MGL bitmap files installed here.
│
├────CURSORS         All MGL mouse cursor files installed here.
│
├────ICONS           All MGL icon files installed here.
│
├────EXAMPLES        Any example source code. Separate sub-directories are
│    ├────MGL        provided for each of the different products.
│    ├────...
│    └────SVGAKIT
│
├────INCLUDE         All installed header files.
│
├────LIB             All installed library files.
│    ├────DOS16      All 16 bit DOS libraries.
│    │    ├───BC3        Borland C++ 3.1
│    │    ├───BC         Borland C++ 4.x
│    │    ├───SC         Symantec C++ 6.x/7.x
│    │    ├───VC         Visual C++ 1.5x
│    │    └───WC         Watcom C++ 10.x
│    ├────DOS32      All 32 bit DOS libraries
│    │    ├───BC         Borland C++ 4.x
│    │    └───WC         Watcom C++ 10.x
│    ├────WIN16      All 16 bit Windows libraries
│    │    ├───BC         Borland C++ 4.x
│    │    ├───SC         Symantec C++ 6.x/7.x
│    │    ├───VC         Visual C++ 1.5x
│    │    └───WC         Watcom C++ 10.x
│    ├────WIN32      All 32 bit Windows libraries
│    │    ├───BC         Borland C++ 4.x
│    │    ├───SC         Symantec C++ 6.x/7.x
│    │    ├───VC         Visual C++ 1.5x
│    │    └───WC         Watcom C++ 10.x
│    └────OS2        All 32 bit OS/2 libraries
│         ├───BC         Borland C++ 2.x
│         ├───CSET       IBM CSet/2
│         ├───HC         Metaware High C++
│         └───WC         Watcom C++ 10.x
│
└────SRC             All source code for building the libraries (if
     ├────MGL        provided).Each package has a separate directory that
     ├────...        contains all of the source code and makefiles for
     └────SVGAKIT    building the library and installing it under
                     \SCITECH\LIB.
```

Using the SuperVGA Kit

This section provides an overview of the SuperVGA Kit, and provides background details on the SuperVGA Kit functionality and how to utilize this functionality in your own applications.

What is the SciTech SuperVGA Kit?

The SciTech SuperVGA Kit is a Software Development Kit for working with VESA VBE compliant SuperVGA graphics cards under both DOS and Windows. This SDK includes full support for all standard VBE devices, as well as devices that support the new VBE/AF Accelerator Functions specification. Although VBE 2.0 or VBE/AF 1.0 is required to get the maximum performance out of this library, it does provide full support for existing controllers that support the VBE 1.2 and lower specifications.

The SciTech SuperVGA Kit provides support for both 16 bit real mode and 32 bit protected mode development under MS-DOS, and both 16 and 32 bit protected mode under Windows 3.1 and Windows '95. Under Windows 3.1 and Windows '95 the SuperVGA Kit is provided as both 16 and 32 bit DLL's, and as staticly linkable libraries. If you wish to use the static link libraries under 16 bit Windows, please read the special section related to this below.

SuperVGA Kit Components

The SuperVGA Kit consists of a number of header files static link libraries for DOS and Windows application, or as 16 or 32 bit DLL's for Windows applications. The following files comprise the standard SuperVGA Kit library package (see installation section for details on where the files will be located):

File	Purpose
SVGA.H	Main SuperVGA Kit header file.
VESAVBE.H	Header file for VESA VBE 1.2/2.0 interface module.
VBEAF.H	Header file for VBE/AF 1.0 interface module.
SVGA.LIB	Static link libraries for default calling conventions.
SSVGA.LIB	Watcom C++ static link libraries for stack calling conventions.
SVGA60.LIB	16 bit SuperVGA Kit for Windows import library.
SVGA60F.LIB	32 bit SuperVGA Kit for Windows import library.
SVGA60.DLL	16 bit SuperVGA Kit for Windows DLL.
SVGA60F.DLL	32 bit SuperVGA Kit for Windows DLL.

Note that in order to use the SuperVGA Kit for Windows, you will also need the WinDirect package (described below) to shut down and restore GDI.

Using Windows 16 bit static link libraries

In order to support 32 bit linear framebuffer access from 16 bit Windows code, the SuperVGA Kit includes special 32 bit linear framebuffer assembler functions. These functions are 32 bit assembler code that reside in a completely separate 32 bit code segment. Before any of the 32 bit code can be run, the SuperVGA Kit code calls a special function in the 32 bit code segment that will convert the selector for that segment to a 32 bit code segment selector.

However current 16 bit C compilers dont know about 32 bit code segments, and by default will automaticaly merge the 32 bit code segment with your applications normal 16 bit code segments. When this happens, the same selector will be used for the 32 bit code segment and any of the merged 16 bit code segments! Naturally the code will not run correctly, and your program will crash.

The solution is to ensure that when you link your application that you tell your linker to turn off the code segment packing option during the link, which will solve the problem. An alternative solution is to always use the provided 16 bit DLL rather than linking with the static link library for 16 bit Windows code.

Building applications with the SuperVGA Kit

To build an application using the SuperVGA Kit, you will need to include the SVGA.H header file in your code, and link your application with both the SVGA.LIB and PMODE.LIB libraries for your DOS Extender (see section of PM/Pro library for more details). You will need to ensure that you select the correct libraries for the specific compiler and calling conventions that you are using. For Widows code you will need to link with the PM/Pro DLL import libraries rather as the PM/Pro library is only provided as DLL's for Windows.

If you are compiling Windows code and wish to use the DLL version of the SuperVGA Kit, you will need to #define SVGA_DLL when you compile all source modules that include the SVGA.H header file, and link with the SuperVGA Kit DLL import libraries for your compiler. The DLL import libraries have the same name as the DLL files for this release of the SuperVGA Kit with the .LIB file extension (i.e. SVGA60.LIB or SVGA60F.LIB for the 6.0 release). Note that for Windows code using the DLL libraries you will not need to link to the PM/Pro library files unless you make explicit calls to these library functions, as the SuperVGA Kit DLL's are already linked to the PM/Pro library DLL's.

The SciTech SuperVGA Kit comes with full source code, so you can also directly include the source code for the SuperVGA Kit in your own programs rather than linking with the pre-compiled libraries. However you will need Turbo Assembler 4.0 or later in order to assemble the provided assembler source code (extract the object files from the provided libraries with your compilers librarian if you don't have Turbo Assembler).

If you don't wish to use the entire SuperVGA Kit libraries in your application but wish to support VBE 2.0 in your applications, we *highly* recommend you link in and use the VESAVBE.C module as your main interface to VBE 2.0. None of the code in

VESAVBE.C is speed critical, and this module takes care of fixing minor bugs in slightly varying VBE implementations to make the high level interface consistent. This module also takes care of isolating protected mode applications from the need to deal with pointers to real mode memory locations by mapping all such data into the default data segment automatically.

Initializing the SuperVGA Kit

Before you can use the SuperVGA Kit, you need to call the SV_init initialization function. This function takes a flag that indicates whether you wish to search for and use VBE/AF services where available, and will return a pointer to the SuperVGA Kit's global device context block. This global device context block maintains all the state information for the SuperVGA Kit, and can be used to directly access many of the SuperVGA Kit's internal variables for maximum performance.

Note that this is slightly different to the previous 5.x releases of the SuperVGA Kit where all global variables were available directly. Since the SuperVGA Kit has been updated to support both 16 and 32 bit DLL's under Windows, the interface was changed to include a global device context block pointer returned to the application. The reason for this is that you cannot share global variables directly in 16 bit DLL's (you can for 32 bit DLL's, but many compilers do not currently support this). The advantage to the new interface is that you can change from the static link libraries to the DLL version of the SuperVGA Kit simply by adding the #define SVGA_DLL and recompiling and linking your code.

Starting a Graphics Mode

Before you can start any graphics mode, you must first find the mode number of the graphics mode that you wish to start. The SuperVGA Kit does not define any standard mode numbers, and it is up to your application code to read through the list of available graphics modes and to search for a graphics mode with the desired resolution and color depth using the VBE_getModeInfo function.

The SuperVGA Kit allows you to start a graphics mode using either the banked framebuffer, a virtual linear framebuffer or a hardware linear framebuffer on the graphics card (depending on the underlying VBE driver's capabilities). When the SuperVGA Kit is initialized, it will interrogate the graphics card and determine if it supports a hardware linear framebuffer. If it does support one, the *linearAddr* variable in the the global device context block will be set to the value of the *physical* address of the linear framebuffer (this is not a CPU address, so don't try to write to it!!). It is then up to your application to pass the *svLinearBuffer* flag to the SV_setMode function when you start the graphics mode, and it will enable the hardware linear framebuffer mode.

If a hardware linear framebuffer mode is not available, the SuperVGA Kit checks to see if it is possible to initialize a virtual linear framebuffer using the 386 virtual memory manager (not possible under DOS boxes and some DOS extenders). If this is possible, it sets the *haveVirtualBuffer* flag in the global device context block to true, indicating that the virtual linear buffer is available. When you initialize a graphics mode, the SuperVGA Kit will check the *useVirtualBuffer* flag you pass to the set mode functions to determine if the virtual linear framebuffer should be used. If this flag is set to true the virtual linear framebuffer mode will be initialized and the *virtualBuffer* flag in the global device context will be set to true to indicate that the virtual buffering mode is currently active.

Once the graphics mode is initialized, you can draw to it using the supplied SuperVGA Kit graphics output functions (not very specactular) or you can write your own high speed rendering code to render directly to the linear framebuffer regions.

Note that if you are using the SuperVGA Kit under Windows, you *must* call the WinDirect WD_startFullscreen function before you initialize a graphics mode, and you *must* call WD_restoreGDI to return back to normal Windows mode. Failing to do so will cause the SuperVGA Kit to write to the display concurrently with GDI and will cause GDI to eventually lock up.

Starting a Virtual Scrolling Graphics Mode

If you wish to use hardware virtual scrolling with the SuperVGA Kit, you must initialize the graphics mode using the SV_setVirtualMode function rather than the usual SV_setMode function. The virtual scrolling version of this function takes two extra parameters that define the vitual height and virtual width of the graphics mode that you wish to initialize, and will attempt to start the graphics mode with the specified virtual dimensions. If the graphics hardware cannot handle a virtual mode of these dimensions, this function will fail.

Once the virutal scrolling mode has been initialised, you can use the SV_setDisplayStart function to set the display start address for the mode, allowing you to scroll the visible window into the framebuffer memory around using the hardware panning registers.

Changing the Color Palette

In order to change the color palette for the current graphics mode, you will need to call the SV_setPalette function. It is important that you use this function to set the hardware palette, as VBE 2.0 and VBE/AF devices may be running in a NonVGA graphics mode in which case directly programming the standard VGA registers is not possible.

Performing Double Buffering

In order to perform double buffering with the SuperVGA Kit, you must pass the *svDoubleBuffer* flag combined with the number of the graphics mode you wish to initialise when you call SV_setMode or SV_setVirtualMode. This flag is necessary to correctly inform the VBE/AF drivers that you wish to perform double buffering, so that the drivers will correctly set aside enough framebuffer memory to hold the double buffers and descrease the size of the available offscreen framebuffer memory used for storing bitmaps etc. If the mode is incapable of double buffering the mode set will still work correctly, however the mode will be set up for single buffered operation.

Once a double buffered graphics mode has been started, the mode will be set up with the active and visible buffers both set to the first buffer in framebuffer memory and the system is essentially in normal single buffered mode. To perform double buffering you simply make calls to the SV_setActiveBuffer and SV_setVisibleBuffer functions to change the active and visible buffers. The active buffer is the buffer where all output will currently be drawn, and the visible buffer is the one that the graphics card is currently displaying. For double buffered animation you set the active buffer to a hidden buffer so that you can draw to it without the user seeing the draw occur on the display, and then switch the visible buffer to the hidden buffer to instantly display the new frame.

If you are doing double buffering and you wish to render directly to the framebuffer, you will need to use the *originOffset* and *bankOffset* variable in the global device context block to determine where the currently active buffer starts. In banked modes these define the starting 64Kb bank and offset with that 64Kb bank for the currently active buffer, while in linear framebuffer modes the *originOffset* variable points directly to the start of the active buffer in the 32 bit linear address space.

Drawing Pixels, Lines and Text

The SuperVGA Kit provides a few limited functions for drawing graphics output, such as pixels, lines and text. These pixel output and line drawing functions are written in highly optimized assembler code so they are very fast. However the primary purpose of the SuperVGA Kit is not to provide a rich set of graphics primitives, so only solid, single pixel lines are supported. The text output function is written in C and calls upon the pixel output functions to draw the text, so it is not particularly fast.

Note that the SuperVGA Kit provides two methods of drawing pixels and lines. The first method is via the standard functions and can be used to draw pixels and lines at any time within your code. The second set of functions are provided for drawing fast *batched* pixels and lines, and provide routines that will draw pixels and lines as fast as possible when you need to draw multiple lots of each different primitive. To use the batched drawing functions, you must call the SV_beginPixel or SV_beginLine functions first, and then call the *fast* variants of the normal drawing functions. When you are finished rendering you must call SV_endPixel or SV_endLine to return to the normal mode of operation.

The batched functions are important for obtaining the maximum performance when drawing lines and pixels on VBE/AF devices, especially when the line drawing is being perfomed in softare (for VBE/AF devices you need to arbitrate direct access to the framebuffer with the GUI engine).

Direct Framebuffer Access

One of the powerful features of the SuperVGA Kit is that it provides your applications with direct access to the hardware video memory framebuffer regions. Access is provided either via a banked framebuffer architecture or via a high performance linear framebuffer archicture on graphics hardware that supports it. This means that you can use the SuperVGA Kit to perform all the low level housekeeping and mode initialization functions required to get up and running in SuperVGA modes, and then use your own high performance rendering code to render directly to the hardware framebuffers.

A banked framebuffer is provided by VBE 1.2 and lower devices and also by VBE 2.0 and VBE/AF devices that don't support a hardware linear framebuffer. In order to access the huge amount of video memory on the graphics card directly, we need to map each section of video memory into a 64Kb *bank* located in the first 1Mb of memory. In order to change the active bank, you need to call provided *SV_setBank* function with the index of the bank that you wish to make active. If you are writing high performance assembler code, you should call the assembler *SV_setBankASM* version of the bank switching function which takes it's parameters in CPU registers rather than on the stack. Once you have changed the active bank to the location where you wish to perform your drawing, you can then draw directly into the framebuffer via *videoMem* pointer provided in the SuperVGA Kit device context block. Note that if you are performing double buffering, the *originOffset* and *bankOffset* variables in the device context block are used to indicate where in video memory the currently active buffer begin, and you will need to offset all your drawing code into the framebuffer using these variables (check the _SVGASDK.ASM module for details on how this is done in assembler code).

If the application is running in a 32 bit protected mode environment, we can simplify access to the video memory on the graphics card by using either a hardware or virtual linear framebuffer. A hardware linear framebuffer allows the application to directly access all of the memory on the graphics card in one contiguous chunk, rather than having to deal with moving a sliding 64Kb window into the framebuffer. A virtual linear framebuffer is enabled by using the 386+ CPU's virtual memory features to create what appears to be a linear framebuffer, but is actually a region of virtual memory that automatically maps the controllers 64Kb banked aperture when the application directly accesses portions of the virtual framebuffer. If a linear framebuffer mode is started with the SV_setMode function, the *videoMem* variable of the global device context block will point to the start of the linear framebuffer region, and you can use this variable to render directly into it. Note that if you are doing double buffering the *originOffset* variable is set to point directly to the start of the currently active buffer in linear framebuffer memory, and the videoMem and *bankOffset* variables are no longer used.

Virtual Linear Framebuffer Access

It is possible that when the graphics mode has been initialized, it has been initialized with a linear framebuffer virtualized in software (if you passed the *useVirtualBuffer* flag to true for the mode set). If you intend to directly render to a virtual linear framebuffer, there are a few caveats that must be understood.

Firstly there is an overhead involved in the page faulting mechanism that is used to automatically map the graphics memory, so it wont be as fast as highly optimized bank switched code (but it comes very close). If you are simply copying blocks of data to the framebuffer in a left to right and top down fashion, the copying will not cause very many page faults and hence the overheads will be small if not negligible. If however you are drawing random lines on the screen, you will find that the overhead involved in the page faulting and bank switching will slow down performance (but it is not that much slower than high speed bank switched assembler code; perhaps 5-10%).

Secondly you simply *cannot* perform a WORD or DWORD memory access that crosses a 64Kb bank boundary. Doing so causes an infinite page fault loop that will hang the machine. The infinite loop occurs because the CPU page faults on the second half of the memory access, which causes a bank switch to occur. Then the *entire* instruction is re-started and causes a memory access to the previous 64Kb bank which has now been mapped out. Another page fault occurs and the instruction is re-started again and we are back where we started.

Hence it is vitally important that any virtual linear framebuffer rendering code ensure that all memory accesses are aligned to either a BYTE, WORD or DWORD boundary to avoid the infinite page fault loop.

Using the VBE/AF Accelerator Functions

The SuperVGA Kit provides transparent support for both VBE 2.0 and below devices, and VBE/AF Accelerator Functions. It will automatically use the VBE/AF graphics accelerator for clearing the framebuffer and drawing lines if the hardware supports that functionality in the current graphics mode (and you set the *useVBEAF* flag to true when the graphics mode was started). However the VBE/AF specification also provides support for a much large array of primitives than just these two functions, such as patterned rectangle fill's, polygon fill's and transparent BitBlt functions.

If a VBE/AF device has been detected and is currently being used by the library, the global variable *AFDC* will be set to a pointer to the VBE/AF device context block for the currently loaded driver. You can then use the supplied VBE./AF 'C' based API in the VBEAF.H header file to directly calls these functions, or you can use the sample assembler code in the _VBEAF.ASM module to write your own high performance accelerated graphics functions.

Note that although we provide a 'C' based API for VBE/AF acceleration in the SuperVGA Kit, the specification passes values to the driver in a combination of registers and parameter blocks that is highly tuned towards calls from tight assembly language loops. For absolute maximum performance in your code, please read the VBE/AF specifications and write your own assembler functions to interface directly with the VBE/AF device driver functions.

In order to allow your acceleration code to run correctly, you must also observe a few rules to ensure correct arbitration between the graphics accelerator hardware and any code that renders directly to the framebuffer. By default when a mode is initialized, it is set up for hardware accelerated rendering and you cannot directly access the framebuffer. Before you do any direct framebuffer access, you must call the SV_beginDirectAccess function to ensure mutual exclusion to the framebuffer, and when you are done call SV_endDirectAccess again to return to normal hardware acceleration mode. If you do not do this, there is a chance that the data stored in the framebuffer will get corrupted, or even worse it is quite possible to lock up the graphics controller hardware requiring a hard reboot.

Porting from the SuperVGA Kit 5.2 release

This release of the SuperVGA Kit is a little different to the previous releases for two reasons. Firstly it now has direct support for both VBE 2.0 and VBE/AF devices, and can be compiled and used as a DLL under Windows. This support does not however come for free, and in order to integrate this support into the SuperVGA Kit we had to modify some of the SuperVGA Kit's API functions.

If you are porting from the 5.2 version of the SuperVGA Kit, the first thing you will notice that you will get hundreds of error messages! Dont dispair, as just about all of the changes that you will need to make can be done with the search and replace functions of your text editor. The biggest change is that the SV_init function now returns a pointer to a global device context block, which must be used to directly access any of the global variables in the SuperVGA Kit (since the variables will be in a different data segment when running with the DLL verisons). Previously the SV_init function used to return a value indicating the VBE version number for the device, so now you must check the VBEVersion variable stored in the device context block that is returned. You will also need to store a copy of this pointer globally so that you code can access it, and you will need to change all reference to the global variables in the SuperVGA Kit to access these variables using the global device context block (check the SVTEST.C modue to see how this has been done). In most cases this can be performed with a number of search and replace operations.

The second change is that the parameters to some of the initialisation functions have been modified and extra ones added. Check the prototypes for any functions that give an error and the changes should be reasonably straight forward.

The third change is that the palette programming functions in the SuperVGA Kit now take all palette values in 8 bits per primary format regardless of what format the hardware supports. If the hardware is currently in 6 bits per primary mode, the palette programming functions will convert the palette values on the fly from the 8 bit format to the required 6 bit format. Hence you will need to change your palette values to be in 6 bits per primary format rather than 8 bits to work with the new palette programming functions.

The fourth change is that the line drawing and pixel plotting functions are now provided in normal and batched mode forms. For maximum performance you should modify you code to use the batched drawing versions whenever possible.

So in summary:

Update the call to SV_init and store the SV_devCtx * globally so you can access the variables in the SuperVGA Kit. You also need to pass a flag indicating if you wish to use VBE/AF services if they are available.

Update all references to the old global variables to reference the variables indirectly via the SV_devCtx*. Most of this is search and replace stuff.

Update calls to SV_setMode to take the new extra parameters.

Modify your palette programming code to pass 8 bits per primary values rather than 6 bits per primary.

Modify your pixel plotting and line drawing code to use the fast batched functions if possible.

Using WinDirect

This section provides details of using WinDirect for directly accessing the video hardware in standard VGA modes, ModeX style tweaked modes and VESA VBE SuperVGA modes.

It also describes how to provide support for user requests to Alt-Tab back to the GDI desktop and then back to your WinDirect application again.

What is WinDirect?

WinDirect is a runtime package for DOS, Windows 3.1 and Windows '95 that provides direct access to the display hardware for both 16 and 32 bit applications. Traditionally Windows applications have had to perform all graphics output using the standard Graphics Device Interface (GDI). Although the GDI is very extensive and powerful, it is also not particularly fast for the sort of graphics that real time applications like interactive video games require.

WinDirect breaks this barrier by allowing high performance applications to shut down the normal GDI interface, and to take over the entire graphics display hardware just like you would normally do under DOS. Once GDI has been shut down, interactive graphics applications can re-program the display controller and write directly to video memory. WinDirect applications can program any standard VGA graphics mode such as 320x200x256, it can re-program the controller and run standard VGA ModeX style graphics, or it can call the standard VESA BIOS services to run high resolution SuperVGA graphics.

WinDirect provides the ability to write a single source application that can run be compiled to run under 32 bit DOS or 32 bit Windows. WinDirect can be used separately with your own high performance graphics code, or can be combined with the SuperVGA Kit to provide portable, fullscreen SuperVGA graphics under both DOS and Windows. WinDirect provides a complete API for managing a fullscreen, unified event queue for mouse and keyboard handling under both DOS and Windows. All mouse and keyboard events are obtained through the WinDirect API and provide discrete events for key down, key up and key repeat events, mouse down, mouse up and mouse movement events.

But best of all, WinDirect applications are still standard Windows applications, so you can use all the standard Windows multi-media API's for digitised sound, CD-ROM audio, networking etc.

WinDirect Components

WinDirect consists of a number of header files and static link libraries for DOS, or as 16 and 32 bit DLL's for Windows applications. The following files comprise the standard WinDirect package (see installation section for details on where the files will be located):

File	Purpose
WDIRECT.H	Standard WinDirect header file
WDIR.LIB	Static link libraries for DOS
SWDIR.LIB	Watcom C++ DOS libraries for stack calling conventions.
WDIR60.LIB	16 bit WinDirect import library
WDIR60F.LIB	32 bit WinDirect import library
WDIR60.DLL	16 bit WinDirect DLL. Contains all 16 bit WD_ functions.
WDIR60F.DLL	32 bit WinDirect DLL. Contains all 32 bit WD_ functions. Requires WDIR60.DLL to interface to 16 bit subsystem code.
DVA.386	Updated virtual framebuffer device for Windows 3.1 from the Video for Windows runtime package. This device driver provides both 16 and 32 bit code with the ability to create a virtual linear framebuffer for SuperVGA devices that do not include a hardware linear framebuffer.

Note also that all the DLL files and DVA.386 must be installed into the standard Windows system directory for correct operation. You must also install the 'device=dva.386' entry in the '[386Enh]' section of the SYSTEM.INI file under Windows 3.1 if you intend to use the framebuffer virtualisation technology. Note also that this is an updated version of DVA.386 that properly supports 32 bit framebuffer virtualisation under Windows 3.1. Previous versions that ship with Video for Windows releases up to 1.1e do not support Win32s applications.

Directly Accessing Standard VGA Modes

The heart of WinDirect is the ability to shut down GDI and start a fullscreen, direct to hardware graphics mode, and then restore GDI back to normal when we are finished. What we are doing essentially is similar to starting a fullscreen DOS box (in fact WinDirect eventually calls the same routines that Windows calls when it swaps between normal GDI graphics and fullscreen DOS sessions). There are two functions related to this, WD_startFullScreen and WD_restoreGDI. These two WinDirect functions take care of all the dirty details, such as calling make the appropriate calls to shut down GDI, ensuring a fullscreen window covers the desktop to capture all events properly and setting up for using the WinDirect event handling API functions.

A simple application that wants to start a full screen standard VGA mode, like mode 13h (320x200x256) would be written as follows:

```
int PASCAL WinMain(HINSTANCE hInst,HINSTANCE hPrev,LPSTR szCmd,int sw)
{
    RMREGS regs;
    uchar  *screenPtr;

    /* Shutdown GDI and start VGA mode 13h */
    WD_startFullScreen(NULL,320,200,false);
    regs.x.ax = 0x13;
    PM_int86(0x10,&regs,&regs);
    WD_inFullscreenMode();

    /* Get pointer to screen memory */
    screenPtr = PM_mapPhysicalAddress(0xA0000,0xFFFF);
```

```
    /* ... do some graphics ... */

    /* Restore GDI and exit */
    WD_restoreGDI();
    return 0;
}
```

Notice that the call to WD_startFullScreen included the resolution of the expected video mode. WD_startFullScreen does not actually start any video modes, but these values are used to scale the mouse coordinates used for the event handling functions and should be equal to the resolution of the video mode you are about to initialise (in this case 320x200x256).

Once the fullscreen mode has been started, Windows will have put the system into a standard VGA video mode. However we want to ensure the system is using the mode we want, so we call the standard VGA BIOS video mode set routine to start mode 13h using the PM_int86 function, which makes calls to the real mode BIOS. The next step it so obtain a pointer to the video memory, which we do using the PM_mapPhysicalAddr function. You could create the pointer using PM_createSelector to make a new selector, but under 32 bit protected mode this would require assembler code to access the video memory. PM_mapPhysicalAddr on the other hand maps the video memory into a 32 bit near pointer for 32 bit protected mode code, so we can directly access the memory from C using normal near pointer arithmetic.

Once you have the pointer to the video memory, you can simply go ahead and blast data straight the screen. See the sample code in WDVGA.C that shows a simple application drawing lines in mode 13h using WinDirect under either DOS or Windows.

Directly Accessing VGA ModeX Modes

Once WinDirect is active in it's fullscreen mode, your application has full control of the video hardware, and in fact you can re-program it just like you would for a normal DOS application. This includes re-programming the standard VGA mode 13h mode into the tweaked ModeX style modes used by many current DOS games, and even programming the graphics accelerator hardware directly.

The only catch is that the normal DOS API functions to program the IO ports may not be available under your Win32 compiler, so you will have to write some small assembler functions (or use inline assembler for your compiler if you wish) to perform normal port IO. Win16 compilers provide the functions so you wont have any problems moving stuff to 16 bit Windows.

Using the SuperVGA Kit with WinDirect

Using the SuperVGA Kit with WinDirect is just as simple as directly programming VGA mode 13h using WinDirect as outline in the example above. A simple application that wants to start a full screen 640x480x256 mode would be written as follows (note that you would *normally* search the VBE mode list for the 640x480x256 mode, but we use mode 0x101 here for brevity):

```
int PASCAL WinMain(HINSTANCE hInst,HINSTANCE hPrev,LPSTR szCmd,int sw)
{
    SV_devCtx   *DC;
    RMREGS      regs;

    /* Init the SuperVGA Kit */
    DC = SV_init(false);
    if (!DC)
        exit(1);

    /* Shutdown GDI and start 640x480x256 */
    WD_startFullScreen(NULL,640,480,false);
    SV_setMode(0x101,true,true);
    WD_inFullscreenMode();

    /* ... do some graphics ... */
    SV_line(0,0,100,100);

    /* Restore text mode, GDI and exit */
    SV_restoreMode();
    WD_restoreGDI();
    return 0;
}
```

See the source code in the WDTEST.C program that shows a fully functional Windows program using the SuperVGA Kit and WinDirect under Windows. This program also properly handles requests to switch back and forth between the GDI desktop and your WinDirect application.

Directly Accessing the VESA BIOS Extensions

Directly accessing the VESA BIOS Extensions (VBE) from WinDirect code is a little more involved that simply setting the video mode. The VBE interface is a real mode software interrupt interface, and expects to be passed real mode addresses to the functions to get the adapter configuration and video mode information blocks. In order to calls these routines you must first allocate a real mode transfer buffer, and indirectly copy all information through this buffer when communicating with the BIOS routines. The PM_allocRealSeg and PM_int86x routines are used to make calls to the VBE routines. A small wrapper function like the following is easy to build and can insulate you from most of the thunking issues:

```
void callVBE(RMREGS *regs, void *buffer, int size)
{
    uint VESABuf_len = 1024;  /* Length of VESABuf             */
    uint VESABuf_sel = 0;     /* Selector for VESABuf          */
    uint VESABuf_off;         /* Offset for VESABuf            */
    uint VESABuf_rseg;        /* Real mode segment of VESABuf */
    uint VESABuf_roff;        /* Real mode offset of VESABuf  */
    RMSREGS sregs;

    if (!VESABuf_sel) {
        if (!PM_allocRealSeg(VESABuf_len, &VESABuf_sel, &VESABuf_off,
                &VESABuf_rseg, &VESABuf_roff))
            exit(1);
        }
```

```
    sregs.es = (ushort)VESABuf_rseg;
    regs->x.di = (ushort)VESABuf_roff;
    PM_memcpyfn(VESABuf_sel, VESABuf_off, buffer, size);
    PM_int86x(0x10, regs, regs, &sregs);
    PM_memcpynf(buffer, VESABuf_sel, VESABuf_off, size);
}
```

Note however that if the information blocks returned from the BIOS contain pointers, those pointers will be real mode pointers, and they will need to be translated to protected mode pointers before you can access the memory they point to. The PM_mapRealPointer function can be used for this.

If you plan to provide direct support for the VESA BIOS extensions in your WinDirect applications without using the SuperVGA Kit, it is highly recommended that you take the VESAVBE.C module from the SuperVGA Kit and use that for interfacing with the VESA BIOS functions.

Interacting with the Mouse and Keyboard

User interaction when running under WinDirect is a little different to the normal type of event processing that Windows application programmers will be used to. Even though WinDirect creates a hidden (hidden because GDI is not visible anymore) window that covers the entire GDI desktop, event handling is not done through a normal Window procedure. Instead WinDirect provides an application event queue which is used to store all the mouse and keyboard events from the user, which can then be queried by WinDirect applications.

Hence WinDirect applications will have an internal event loop that will involve reading the WinDirect event queue with either WD_peekEvent or WD_getEvent, and then processing the events with a large switch statement similar to a standard Window procedure. The WinDirect event queue is not just for mouse and keyboard events, but can be used for storing application specific events and timer tick events.

Also note that the WinDirect libraries for DOS provide an identical event queue handling system for keyboard and mouse events, which provides a much more powerful way of handling the mouse and keyboard under DOS than the standard DOS interface functions normally do. WinDirect provides full support for key up, key down and key repeat events while the DOS services only provide support for key down events.

Displaying a Mouse Cursor in WinDirect Modes

WinDirect does not directly handle the drawing of the mouse cursor when in fullscreen video modes (under both DOS and Windows), since it has no idea what video mode the application will put the system into. WinDirect does however allow you to register a callback that will be called whenever the mouse is moved, allowing your application code to include custom mouse drawing routines for displaying a mouse cursor when in fullscreen video modes. When you register your mouse callback code with the WD_setMouseCallback function, WinDirect will call your supplied callback function whenever the mouse moves, and will pass it the current mouse cursor coordinates, properly scaled to the resolution that you originally specified in your call to WD_startFullScreen.

Note that it is up to your application to determine how and when to draw the mouse cursor, and to ensure that the cursor is hidden when drawing graphics output to the display to avoid drawing to the same portion of the graphics display.

Note also that this routine is currently not asynchronous, but will only be called while your application is calling the WinDirect event handling functions. If your application does a significant amount of processing without checking the event queue, the mouse cursor will appear to freeze while this is occuring. Under Windows '95 however you could perform the complex processing in a separate thread, allowing both threads to proceed asynchronously.

Switching back to the Normal GDI Desktop

When a WinDirect application is active, other Windows applications cannot send output to the display, and will be blocked by Windows while your application is active. WinDirect will also not allow the user to Alt-Tab or Alt-Esc back to other running Windows applications without assistance from your application code. If you wish to allow the user to switch back to the desktop, you will need register a suspend application callback with WinDirect using the WD_setSuspendAppCallback function.

When WinDirect detects that the user wishes to switch back to GDI, WinDirect will call your callback. It is then up to your callback code to save the current state of the application so that it can be restored at a later date. This may include saving the entire contents of video memory either to disk or a memory buffer if you application cannot re-build the video memory contents dynamically. When the user is ready to switch back to your WinDirect application, WinDirect will call your callback code once again and request that it restore the system back to the state it was in. This includes restoring the active video mode, as WinDirect will not save and restore the video mode when switching to GDI and back.

Note that the WinDirect libraries for DOS provide the suspend application callback API, however these functions do nothing under DOS and your callback functions will never be called.

NOTE: It is important that you pass a handle to your applications main window if your application uses a normal GDI main window, so that WinDirect can properly minimize your application when it switches back to GDI mode. If you dont have a main window (your application goes direct to fullscreen mode) then you can pass a NULL in this parameter.

Debugging WinDirect Applications

In order to debug WinDirect applications under Windows 3.1 or Windows '95, you must run your normal debugger in dual monitor mode, with all debugger output displayed on a monochrome monitor, or you must use remove debugging. Using a normal fullscreen or

GDI debugger window, you will not be able to see the standard Windows debuggers output screens once you have shut down GDI.

Once you have set up your debugging environment properly, you should be able to step through and trace all WinDirect code while GDI is still shut down. If your application terminates abnormally while in full screen mode, you can simply reset the application to unload the WinDirect DLL's, and this will cause normal GDI operation to be restored.

The WDBUG.EXE debugging applet

To assist in the debugging process and to be able to properly restore GDI mode if an errant WinDirect application has hung the system, we have provided a small debugging applet. To use the applet you simply need to run the applet before you run any other WinDirect applications (or put it into your Windows startup folder).

Once loaded the applet will perform a GDI shutdown and restore when you hit the Alt-R key combination. Hence if your application has hung while in fullscreen mode, you can simply hit Alt-R to restore GDI mode and kill off the offending application rather than having to reboot the system.

Using AVIDirect

This section provides an overview of AVIDirect, and provides background details on AVIDirect's functionality and how to utilize this functionality in your own applications.

What is AVIDirect?

AVIDirect is a simple, high performance module for playing back 8,15,16 and 24 bit DIB bitmaps from system memory directly to fullscreen modes using WinDirect. AVIDirect supports playback in native 320x200 and 320x240 modes where possible and uses hardware page flipping to toally eliminate any tearing artifacts during video playback. For maximum performance on low end machines, AVIDirect can be configured to allow the code to decode directly to video memory, however some tearing effects may be noticeable during fast changing scenes. AVIDirect supports playback in any 8,15,16 and 24 bits per pixel video modes and provides support for automatically performing a 1x2 zoom or a 2x2 zoom for playing back 320x200 or 320x240 images into 320x400 or 320x480 graphics modes or 640x400 and 640x480 graphics modes.

Playing back AVI files is as simple as installing the supplied Video for Windows fullscreen draw handler and using standard MCI AVI video playback function calls to play back your videos. For more control or if you are using your own internal codec's, you can also call the AVIDirect API directly to initialize a graphics mode, and to blit buffers from system memory to video memory, or decode directly to video memory. Note that if you are directly calling the AVIDirect API, 24 bit DIB's can be played back in any 15,16 or 24 bit graphics mode, while 8, 15 and 16 bit native DIB's can only be played back in the corresponding native mode. Hence if your video playback modules can only decompress to 24 bit then you can still play back fast in 15 or 16 bit video modes. However if you can decompress to real 8, 15 or 16 bit DIB's you can blt these directly to the display for maximum performance.

NOTE: Because this library uses hardware linear and virtual linear framebuffer code, it requires that VBE 2.0 or higher be installed.

AVIDirect Components

AVIDirect consists of a number of header files, DLL's and import libraries for 16 and 32 bit for Windows applications. The following files comprise the standard AVIDirect package (see installation section for details on where the files will be located):

File	Purpose
AVIDIREC.H	Standard WinDirect header file
AVDIR60.LIB	16 bit AVIDirect import library
AVDIR60F.LIB	32 bit AVIDirect import library
AVDIR60.DLL	16 bit AVIDirect DLL. (Windows 3.1 and Windows '95)
AVDIR60F.DLL	32 bit AVIDirect DLL (Windows '95 only).

Note that AVIDirect requires the WinDirect and PM/Pro library DLL's to be installed in order to function correctly. It does not use the SuperVGA Kit but contains it's own built in functionality.

Playing an AVI File in Fullscreen Modes

Please see the sample AVIPLAY.C source code that shows how to play back AVI files directly using AVIDirect.

Using the PM/Pro Library

This section provides an overview of the PM/Pro Library, and provides background details on the PM/Pro Library's functionality and how to utilize this functionality in your own applications.

What is the PM/Pro Library?

The PM/Pro library provides a small, DOS extender independant API for protected mode programming for both DOS and Windows applications. It covers all of the issues that usually burden the programmer converting their code to work under protected mode such

as issuing real mode interrupts, calling real mode code, allocating real mode memory, directly accessing the low 1Mb of real mode memory and installing protected mode interrupt handlers (DOS only).

All SciTech Software products use the PM/Pro libraries to provide support for the various DOS extenders and for fullscreen Windows support. Changing DOS extenders is simply a matter of linking with a different version of the PM/Pro library. It fully supports real mode, 16 bit protected mode and 32 bit protected mode programming under DOS, Windows 3.1 and Windows '95. The PM/Pro libraries are fully DPMI compliant, so any code written with these libraries should run without problems under a Windows 3.1, Windows '95, Windows NT or OS/2 2.x DOS box. Applications written to use the Windows version of the PM/Pro library will run under Windows 3.1 and Windows '95 but not under Windows NT.

As well as providing pre-built selectors for the BIOS data area and VGA frame buffer areas, the PM/Pro library also provides support for creating near pointers to physical memory locations. This includes memory below the DOS 1Mb mark (such as the VGA framebuffer) and also high extended physical memory locations such as the hardware linear framebuffer for SuperVGA graphics cards. The method used to map these memory locations is fully DPMI compliant, and works properly under Windows and OS/2 DOS boxes. Of course this is only available to 32 bit protected mode applications, but it does provide the absolute fastest way to access physical memory locations.

In order to be able to support graphics cards that don't have a hardware linear framebuffer, the PM/Pro library provides support for creating a 'virtual' linear framebuffer. A virtual linear framebuffer uses the 386+ memory mapping facilities to change SuperVGA banks using a page fault handler, rather than explicitly writing code to handle this. For maximum performance we hook the page fault handler directly from the Interrupt Descriptor Table (IDT) so we can handle our page faults as quickly as possible.

The virtual linear framebuffer support is currently only supported under the DOS4G/W and PMODE/W DOS extenders, and for 16 and 32 bit applications running under Windows 3.1 and Windows '95. Because implementing this for the DOS libraries requires ring 0 access, this is not compatible with Windows and OS/2 DOS boxes. You can use VF_available() to determine if the support is available and it will return false if the environment will not allow it. Under Windows 3.1 this support is available if the supplied DVA.386 VxD is installed, and is always provided by the built in VFLATD.386 VxD in Windows '95.

PM/Pro Library Components

The PM/Pro library consists of a number of header files and static link libraries for DOS, or as 16 and 32 bit DLL's for Windows applications. The following files comprise the standard PM/Pro package (see installation section for details on where the files will be located):

File	Purpose
PMODE.H	Standard PM/Pro header file
PMPRO.H	Standard PM/Pro header file (interrupt handling etc)
PMODE.LIB	Static link libraries for DOS
SPMODE.LIB	Watcom C++ DOS libraries for stack calling conventions
PMPRO60.LIB	16 bit PM/Pro import library
PMPRO60F.LIB	32 bit PM/Pro import library
PMPRO60.DLL	16 bit PM/Pro library DLL. Provides all the 16 bit PM_ functions for calling real mode interrupts etc.
PMPRO60F.DLL	32 bit PM/Pro library DLL. Provides all the 32 bit PM_ functions for calling real mode interrupts etc. Requires PMPRO60.DLL to interface to 16 bit subsystem components.

Note that all the DLL files must be installed into the standard Windows system directory for correct operation.

Issuing Real Mode Interrupts

In order to issue real mode interrupts under protected mode DOS and Windows, you must use your DOS extender or the DPMI interface routines to simulate the real mode interrupts. The catch however is that real mode code cannot be called directly from protected mode DOS or Windows code, so PM/Pro provides an API that can be used for issuing real mode interrupts. The API is identical to the standard DOS int86() functions, so you can pretty much port most code directly from real mode DOS by changing the int86() functions to PM_int86 functions and changing the names of the register structures used.

The other catch is that when communicating with real mode code like the VESA BIOS, any parameters blocks to be passed in must be located in real mode memory, and you must pass real mode addressed to the functions. To get around this the PM/Pro library provides the routines PM_allocRealSeg and PM_mapRealPointer for allocating and directly accessing real mode memory.

Calling Real Mode Code

As well as issuing software interrupts, you may need to directly call real mode code (like BIOS code or TSR code) from your application. PM/Pro provides the PM_callRealMode function to call a real mode far function. The API is very similar to PM_int86, except that the function being called ends in a 16 bit far return statement rather than a return from interrupt statement.

Mapping Physical Memory Locations

In order to directly access the video memory on a graphics card, you need to be able to map the physical memory where the framebuffer contents is located into the applications address space. The PM/Pro library provides the PM_mapPhysicalAddr function, which will map any physical memory location into a native mode pointer for the application. For 16 bit applications this will be a far pointer, but for 32 bit applications this will be a 32 bit near pointer for maximum speed.

PM_mapPhysicalAddr will map physical memory locations both below and above the 1Mb memory boundary, but cannot map a region of memory that cross the 1Mb boundary. This function can also map memory blocks larger than 64Kb for both 16 and 32 bit applications, so you can map the entire linear framebuffer for an advanced SuperVGA graphics card directly into the applications code space. For 16 bit Windows code, you will need to write some special 32 bit assembler functions to access this memory block as you will need to use offsets that are larger than 64Kb. See the SuperVGA Kit source code which includes 32 bit linear framebuffer code that is callable from 16 bit Windows.

Virtualizing the SuperVGA Framebuffer

Many older SuperVGA devices do not include hardware linear framebuffer support, and hence access to the framebuffer on these devices must be done through a small 64Kb bank switched window. By being able to access the entire framebuffer via a 32 bit near pointer, you can use the same code for rendering to a system memory buffer and for rendering directly to video memory (hence enabling you to write one set of code that can be used for drawing to a DIB and blting to a real GDI window, or rendering directly to the framebuffer for fullscreen WinDirect code). However the bank switched architecture prevents this and special case code must be written for rendering to such a non-linear buffer device.

The PM/Pro library also provides a set of functions for creating a virtual linear framebuffer for such devices by using the 386 paging system to automatically handle the SuperVGA bank switching. All you need to virtualise the framebuffer is a 32 bit protected mode bank switch routine that can be relocated into the code space of the virtual framebuffer device driver (DVA.386 for Windows 3.1, VFLATD.386 for Windows ‘95, or our special VFLAT code under DOS). The virtual buffer device can handle graphics cards with either 4Kb banks or 64Kb banks, which should support all currently existing graphics cards.

Using the Zen Timer

This section provides an overview of the Zen Timer Library, and provides background details on the Zen Timer Library's functionality and how to utilize this functionality in your own applications.

What is the Zen Timer?

The Zen Timer is a 'C' callable library for timing code fragments with microsecond accuracy (less under Windows). The code was originally developed by Michael Abrash for his book "Zen of Assembly language - Volume I, Knowledge" and later in his book "Zen of Code Optimization". We modified the code and made it into a 'C' callable library and added a few extra utility routines, the ability to read the current state of the timer and keep it running, and added a set of C++ wrapper classes. We also added a new Ultra Long Period timer that can be used to time code that takes up to 24 hours to between calls to the timer and with an accuracy of 1/10th of a second.

This library supports real mode, 16 bit protected mode and 32 bit protected mode under DOS, Windows 3.1, Windows NT, Windows '95. Under the Windows environments the Zen Timer library maps the calls to standard Window's API functions that will provide the highest accuracy timing (not quite the same accuracy as under DOS).

Zen Timer Library Components

The Zen Timer Library consists of a number of header files and static link libraries for DOS and Windows. The following files comprise the standard Zen Timer Library package (see installation section for details on where the files will be located):

File	Purpose
ZTIMER.H	Standard Zen Timer Library header file
ZTIMER.LIB	Static link libraries for DOS and Windows
SZTIMER.LIB	Watcom C++ DOS libraries for stack calling conventions

Timing with the Long Period Zen Timer

Before you can use the Zen Timer in your code, you must first always call the ZTimerInit function to initialize the Zen Timer Library. Once you have done this, to use the timer, isolate the piece of code you wish to time and bracket it with calls to LZTimerOn and LZTimerOff. You then call LZTimerCount to obtain the count use it from within your C program. For example:

```
int i;
```

```
void main(void)
{
    ulong count;

    ZTimerInit()
    LZTimerOn();
    for (i = 0; i < 20000; i++)
        i = i;                          /* Do some work */
    LZTimerOff();
    count = LZTimerCount();
}
```

While the timer is running, you can call the LZTimerLap function to return the current count without stopping the timer from running.

One point to note when using the long period time however, interrupts are ON while this timer executes. This means that every time you hit a key or move the mouse, the timed count will be longer that normal. Thus you should avoid hitting any keys or moving the mouse while timing code fragments if you want highly accurate results. It is also a good idea to insert a delay of about 1-2 seconds before turning the long period timer on if a key has just been pressed by the user (this includes the return key used to start the program from the command line!). Otherwise you may measure the time taken by the keyboard ISR to process the upstroke of the key that was just pressed.

Note that under DOS the Long Period Zen Timer has a cumulative limit of approximately 1 hour and 10 mins between calls to LZTimerOn and LZTimerOff.

Timing with the Ultra Long Period Zen Timer

As well as the normal long period Zen Timer functions, we also provide functions that implement an Ultra Long Period Zen Timer. This version of the timer has lower accuracy and can time intervals that take up to 24 hours to execute. There are two routines that are used to accomplish this; ULZReadTime() and ULZElapsedTime(). The way to use these routines is simple:

```
void main(void)
{
    ulong   start,finish,time;

    ZTimerInit()
    start = ULZReadTime();

    /* Do something useful in here */

    finish = ULZReadTime();
    time = ULZElapsedTime(start,finish);
}
```

Calling ULZReadTime latches the current timer count and returns it. You call ULZElapsedTime to compute the time difference between the start and finishing times, which is returned in 1/10ths of a second. If you are using C++, you may want to use the simpler C++ classes, which have a common interface for all timers.

When using the Ultra Long Period timer class you must ensure that no more than 24 hours elapses between calls to start() and stop() or you will get invalid results. There is no way that we can reliably detect this so the timer will quietly give you a value that is much less than it should be. However, the total cumulative limit for this timer is about 119,000 hours which should be enough for most practial purposes, but you must ensure that no more than 24 hours elapses between calls to start() and stop(). If you wish to use the timer for applications like ray tracing, then latching the timer after every 10 scanlines or so should ensure that this criteria is met.

Using the C++ interface

If you are using C++, you can use the C++ wrapper classes that provide a simpler and common interface to all of the timing routines. There are two classes that are used for this:

LZTimer	C++ Class to access the Long Period Zen Timer
ULZTimer	C++ Class to access the Ultra Long Period Zen Timer

Each class provides the following member functions:

start() member function

The start() member function is called to start the timer counting. It does not modify the internal state of the timer at all.

lap() member function

The lap() member function returns the current count since the timer was started. This count is the total amount of time that the timer has been running since the last cll to reset() or restart(), so it is cumulative. The lap() member function does not stop the timer, nor does it change the internal state of the timer.

stop() member function

The stop() member function is called to stop the timer from counting and to update the internal timer count. The internal timer count is the total amount of time that the timer has been running since the last call to reset() or restart() so it is cumulative.

reset() member function

The reset() member function resets the internal state of the timer to a zero count and no overflow. This should be called to zero the state of the timer before timing a piece of code.

Note that the reset operation is performed every time that a new instance of one of the timer classes is created.

restart() member function

The restart() member function simply resets the timers internal state to a zero count and begins timing.

count() member function

The count() member function returns the current timer count, which will be in fractions of a second. You can use the resolution() member function to determine how many seconds there are in a count so you can convert it to a meaningful value. Use this routine if you wish to manipulate and display the count yourself. If the timer has overflowed while it was timing, this member function will return a count of 0xFFFFFFFF (-1 long).

overflow() member function

The overflow() member function will return true if the timer has overflowed while it was counting.

resolution() member function

The resolution() member function returns the number of seconds in a timer count, so you can convert the count returned by the count() member function to a time in seconds (or minutes, or whatever). The value returned is a floating point number, which simplifies the conversion process.

operator << () friend function

This a convenience function that outputs a formatted string to a C++ output stream that represents the value of the internal timer count in seconds. The string represents the time to the best accuracy possible with the timer being used.

Developing for Maximum Compatibility

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

Dont Assume A0000h for the Banked Framebuffer Address

Cards Affected:
Number Nine Imagine 128 series I

Because SuperVGA graphics has it's origins in the original VGA standard and the framebuffer for VGA graphics modes was always located at A000:0000h, many programmers assume that the banked framebuffer aperature for SVGA graphics modes is always located at A000:0000h. This is not always true, and it is possible for some cards to map the banked framebuffer to a different address.

Hence you should always check the value that the VESA interface returns for the base address for the banked framebuffer for every graphics modes that you use.

Check if VGA Compatible Before Touching Any VGA Registers

Cards Affected:
ATI Mach32 and Mach64
Diamond Viper series (Weitek P9000 and P9100)
Diamond Edge 3D (NVidia NV1)
IBM XGA
IIT AGX
Matrox Millenium
Number Nine Imagine 128
Most newer cards

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

If the graphics controller is based on a NonVGA graphics hardware technology (and many popular ones are), in the SuperVGA graphics modes the VGA registers simply do not exist anymore, and attempting to synch to these registers will put your code into an infinite loop.

So be forewarned that doing any fiddling with the standard VGA registers is asking for trouble on certain graphics card that use NonVGA controllers to program the SuperVGA graphics modes (and lots more of these are coming out).

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

Check if VGA Compatible Before Directly Programming the DAC

Cards Affected:

ATI Mach32 and Mach64
Diamond Viper series (Weitek P9000 and P9100)
Diamond Edge 3D (NVidia NV1)
IBM XGA
IIT AGX
Matrox Millenium
Number Nine Imagine 128
Most newer cards

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

Hence if you need to program the color palette, you should always try to use the supplied VBE 2.0 and VBE/AF palette programming routines rather than programming the palette directly. If you must have your own palette programming code, make sure you check the NonVGA attribute bit as discussed above, and if a NonVGA mode is detect you will have to program the palette via the standard VBE 2.0 or VBE/AF services.

Handling Graphics Cards with Only Memory Mapped Registers

Cards Affected:

Alliance ProMotion 3210/6410
Number Nine Imagine 128
Some newer cards

Once area that is not very well covered in the VBE 2.0 specs is support for controllers that only have memory mapped registers rather than IO mapped registers. For these controllers there are a number of small issues that need to be handled in order to make sure that the 32 bit VBE 2.0 relocateable functions work correctly. If order for these functions to work, a selector to the memory mapped registers must be passed in the ES register to the bank switching and display start address programming routines, and in the DS register for the palette programming routine. If this selector is not passed, the code will not be able to correctly access the necessary memory mapped registers and your application will probably crash.

Note that the UniVBE and our UVBELib device support libraries have some special case code that will work correctly if DOS4GW is used, but not with any other DOS extender. However you should make sure you code correctly handles these situations, as it may end up running on a different VBE implementation that does not have special case code for DOS4GW.

Please look at the code in the SVGASDK.C and _SVGASDK.ASM source files in the SuperVGA Kit that perform this function (VBE20_setBankAES etc). It is possible to handle these situations correctly without impacting the performance on controllers where this is not an issue. This is also an issue that needs to be solved now, because there are a number of new graphics controllers coming out that have only memory mapped registers.

Provide for Solid Backwards Compatibility

If you are developing your application to take advantage of the latest VBE 2.0 standards, you should ensure that you all provide a good set of compatability fallbacks for your appliation. There will be cases in the field where your customer may not be able to get a proper VBE 2.0 driver running on their system, and may not be able to get even a VBE 1.2 driver working properly. Hence you should always provide support for at least a standard VGA mode if possible (Mode 13h or ModeX will suffice) or VBE 1.2 support. If you are developing an application that runs in only SuperVGA modes (640x480 and above) then you should at least ensure that your application runs properly on systems with only VBE 1.2 drivers installed.

Although the performance will not be nearly as great with VBE 1.2, a customer is less likeing to be raving mad when they call your tech support lines if the game at least runs. Once they have the game running and wish to get more performance, they will spend more time seeking out higher performance drivers, or will eventually upgrade their graphics card.

Dont Assume all SVGA Low Res Modes are Available

Also note that on some systems, high performance low resolution graphics modes are not always available, so you should not develop your game to rely on the presence of these modes. On some systems modes below 512x384 are not available, so the only available low resolution modes may be the standard VGA Mode 13h and ModeX modes. Hence if you wish to use low resolution, high performance graphics modes you should always check to see if the mode are available, and provide options for the user to select other modes that may be available (on some systems 200/240 high modes are not available, but 400/480 line modes are).

Note that systems that do not support high performance SVGA low resolution modes are few and far between (less than 5% of the installed base), but you should ensure that your code is ready to handle situations where the exact modes that you want are not available.

Develop for the Future with Scalability

An important criteria for developing a successfull application is to attempt to obtain maximum performance across a variety of target hardware systems. You should develop your applications to be as scalable as possible, both in terms of the resolutions and color depths that are supported. If you can get your game to run in 320x200x256 linear framebuffer mode, this will probably provide the absolute maximum performance and compatability in the field. However customers with high end systems will be wanting to run your games at higher resolutions and color depths if possible. Hence you should also develop your games to be fully scalable in terms of resolutions and color depths if possible. Even though the performance may not be so great at 640x480x256 on present day system's, a year from the time that your game is released is may well be possible to support this mode with enough speed to run your game.

If you are developing a 3D game that relies heavily on texture mapping and detailed 3D worlds, you should consider developing the game with multiple levels of detail for the world and the textures. This will allow customers with lower performance machines (like 486/66 VLB systems) to be able to tune the details of the game down to increase performance. Customers with high performance systems or with systems that will ship after your game has been completed can crank up the details and resolution to get a richer game playing experience.

Include an Option for Rendering to a System Buffer

Cards Affected:
Diamond Viper series (Weitek P9000 and P9100)

One of the main reasons for having an option to render to a system buffer is for compatibility. It fixes two problems: 1.) Some cards cannot double buffer, so you will only get one page of video RAM when you query the card. For example, cards based on the Weitek P9x000 chips (like the Diamond Viper) only support a single VBE buffer in many modes. In order to make your software compatible with the Diamond Viper, you need to have an option to render into system memory. 2.) It will allow you to support cards that may not have enough video memory to properly double buffer in the modes that you need. This gives you a fall back so the user with less than the required VRAM can still run your application.

Use Virtual Linear Frame Buffer Services

Cards Affected:
Compaq Qvision 2000
Matrox MGA Impression series

You should strongly consider supporting a virtual linear frame buffer in your code. The SuperVGA Kit includes code for creating a virtual linear frame buffer using DOS4GW or Windows 3.1/95. This will enable you to support cards that don't have a hardware linear frame buffer (or it's broken) and it will allow you to support cards such as the Matrox MGA/Compaq Qvision 2000 cards that only support a 4k bank and no linear frame buffer modes. A virtual linear buffer will free you from the need to support each bank size in your main code.

Reference Section

This section provides a detailed reference of all functions, data structures and global variables in the SuperVGA Kit and associated libraries

Function Reference

The following is a detailed reference of all functions in the SuperVGA Kit and associated libraries.

Reference Entry Template

Summary of what the function does.

Syntax

```
<type> function( <type> parameter[,...] )
```

Prototype in

header.h

This lists the header file(s) containing the prototype for the function. The prototype of a function may be contained in more than one header file, in which case all the files would be listed, so use whichever one is more appropriate.

Parameters

Briefly describes each of the function parameters.

Return value

This section describes the value returned by the function (if any).

Description

This section describes what the function does, the parameters it takes and any details you might need to know in order to get full use out of the function.

See also

This section gives a list of other related functions in the library that may be of interest.

LZTimerCount

Returns the current count for the Long Period Zen Timer.

Syntax

```
ulong LZTimerCount(void);
```

Prototype in
ztimer.h

Return value
Count that has elapsed in microseconds.

Description
Returns the current count hat has elapsed between calls to LZTimerOn and LZTimerOff in microseconds.

See also
LZTimerOn, LZTimerOff

LZTimerLap

Returns the current count for the Long Period Zen Timer and keeps it running.

Syntax
```
ulong LZTimerLap(void);
```

Prototype in
ztimer.h

Return value
Count that has elapsed in microseconds.

Description
Returns the current count hat has elapsed since the last call to LZTimerOn in microseconds. The time continues to run after this function is called so you can call this function repeatedly.

See also
LZTimerOn, LZTimerOff

LZTimerOff

Stops the Long Period Zen Timer counting.

Syntax
```
void LZTimerOff(void);
```

Prototype in
ztimer.h

Description
Stops the Long Period Zen Timer counting and latches the count. Once you have stopped the timer you can read the count with LZTimerCount. If you need highly accurate timing,

you should use the on and off functions rather than the lap function as the lap function does not subtracte the overhead of the function calls from the timed count.

See also
LZTimerOn, LZTimerCount

LZTimerOn

Starts the Long Period Zen Timer counting.

Syntax
```
void LZTimerOn(void);
```

Prototype in
ztimer.h

Description
Starts the Long Period Zen Timer counting. Once you have started the timer, you can stop it with LZTimerOff or you can latch the current count with LZTimerLap.

See also
LZTimerOff, LZTimerLap

PM_allocRealSeg

Allocate a block of real mode memory.

Syntax
```
int PM_allocRealSeg(uint size,uint *sel,uint *off,uint
*r_seg,uint *r_off);
```

Prototype in
pmode.h

Parameters

size	Size of block in bytes to allocate
sel	Protected mode selector for memory block
off	Protected mode offset for memory block
r_seg	Real mode segment for memory block
r_off	Real mode offset for memory block

Return value
1 on success, 0 on failure.

Description
Allocates a block of real mode memory, and returns both the protected mode address (selector and offset) and the real mode address of the memory block. The memory block

can be a maximum of 64Kb in length, and will always be allocated below the 1Mb memory boundary.

Note that real mode memory is a scarce resource, so you should try to allocated as little real mode memory as possible and ensure that you free it up when you are done. If at all possible, you should try to re-use the same real mode memory block as a single transfer buffer for communicating with real mode code.

See also
PM_freeRealSeg

PM_callRealMode

Call a real mode function.

Syntax
```
void PM_callRealMode(uint seg,uint off, RMREGS *regs,
RMSREGS *sregs)
```

Prototype in
pmode.h

Parameters

seg	Segment of real mode function to call
off	Offset of real mode function to call
regs	RMREGS structure containing general CPU register values
sregs	RMSREGS structure containing real mode segment register values

Description
Calls a real mode *far* function located at the specified real mode segment and offset address. Before calling the function, the values stored in the *regs* and *sregs* structures will be loaded into the CPU general purpose registers and real mode segment registers before the real mode function is called. After the function has returned, the state of the CPU register will be saved the in the *regs* and *sregs* structures.

Note that this routine calls a *real mode* function, and does not do any translation on parameters passed to the real mode code. The real mode code will be expecting any addresses passed in to be proper real mode addresses for memory located in the first 1Mb of real mode memory. You will need to allocate a real mode transfer buffer using PM_allocRealSeg and copy the information temporarily into this transfer buffer. You cannot simply pass normal protected mode addresses to the real mode code.

Note that the real mode function must end with a far return.

See also
PM_int86, PM_int86x

PM_createCode32Alias

Create a 32 bit code segment alias (16 bit Windows only)

Syntax

```
uint PM_createCode32Alias(uint sel);
```

Prototype in

pmode.h

Parameters

sel	16 bit selector to alias

Return value

Newly allocated 32 bit code segment selector, or 0 on failure.

Description

Creates a 32 bit code segment selector with the same base address and limit as the original 16 bit selector. This can be used to create a 32 bit code segment for executing 32 bit protected mode code from within a 16 bit Windows application.

PM_createSelector

Creates a protected mode selector.

Syntax

```
uint PM_createSelector(ulong base,ulong limit);
```

Prototype in

pmode.h

Parameters

base	Physical base address for selector (in bytes)
limit	Limit for selector (in bytes; 64Kb is a limit of 0xFFFF)

Return value

Newly allocated selector, or 0 on failure.

Description

Creates a protected mode selector given the specified base and limit. For 16 bit protected mode applications, the limit for this selector can only be a maximum of 64Kb in length. For 32 bit applications and for 16 bit Windows applications running in 386 enhanced mode, the length may be any value up to 4Gb.

See also

PM_freeSelector, PM_getByte, PM_getWord, PM_getLong, PM_setByte, PM_setWord, PM_setLong

PM_freeRealSeg

Frees a real mode memory block.

Syntax

```
void PM_freeRealSeg(uint sel,uint off);
```

Prototype in

pmode.h

Parameters

sel	Selector for memory block to free
off	Offset for memory block to free

Description

Frees a real mode memory block previously allocated with PM_allocRealSeg. Note that real mode memory is a scarce resource, so you should try to allocated as little real mode memory as possible and ensure that you free it up when you are done. If at all possible, you should try to re-use the same real mode memory block as a single transfer buffer for communicating with real mode code.

See also

PM_allocRealSeg

PM_freeSelector

Frees a protected mode selector.

Syntax

```
void PM_freeSelector(uint sel);
```

Prototype in

pmode.h

Parameters

sel	Protected mode selector to free

Description

Frees a protected mode selector previously allocated with PM_createSelector.

See also

PM_createSelector

PM_getBIOSSelector

Returns a selector to the BIOS data area.

Syntax
```
uint  PMAPI PM_getBIOSSelector(void);
```

Prototype in
pmode.h

Return value
Selector to BIOS data area, or 0 on failure.

Description
Returns a selector to the BIOS data area, normally located at segment 0x40 in real mode memory. This is a pre-built selector and will be re-used by all subsequent calls to this routine. If you need to access the BIOS data area, you should call this routine to get a selector to this segment.

This function is also valid for 32 bit protected mode applications (including Win32), however you need to access the memory using a far pointer. Most 32 bit compilers do not support this, but you can use the PM_getByte/PM_setByte family of functions to access his memory if speed is not critical (otherwise you will have to write the code in assembler).

See also
PM_createSelector, PM_freeSelector, PM_getVGASelector, PM_getByte, PM_getWord, PM_getLong, PM_setByte, PM_setWord, PM_setLong

PM_getByte

Reads a byte from a far address.

Syntax
```
uchar PM_getByte(uint s, uint o);
```

Prototype in
pmode.h

Parameters
s	Selector for byte to read
o	Offset of byte to read

Return value
Byte read from the specified location.

Description
Reads a byte from a far address. In 16 bit protected mode you can simply access the memory using a normal far pointer, however in 32 bit protected mode, most compilers do not directly support 48 bit far pointers (16 bit selector, 32 bit offset). This function provides the ability to read memory using a different selector to the default DS selector in 32 bit protected modes.

For speed critical code, you should write the code directly in 32 bit assembler.

See also
PM_getWord, PM_getLong, PM_setByte, PM_setWord, PM_setLong, PM_memcpynf, PM_memcpyfn

PM_getLong

Reads a 32 bit long from a far address.

Syntax
```
ulong PM_getLong(uint s, uint o);
```

Prototype in
pmode.h

Parameters
s	Selector for long to read
o	Offset of long to read

Return value
Long read from the specified location.

Description
Reads a long from a far address. In 16 bit protected mode you can simply access the memory using a normal far pointer, however in 32 bit protected mode, most compilers do not directly support 48 bit far pointers (16 bit selector, 32 bit offset). This function provides the ability to read memory using a different selector to the default DS selector in 32 bit protected modes.

For speed critical code, you should write the code directly in 32 bit assembler.

See also
PM_getByte, PM_getWord, PM_setByte, PM_setWord, PM_setLong, PM_memcpynf, PM_memcpyfn

PM_getModeType

Returns the current operating system mode.

Syntax
```
int PM_getModeType(void);
```

Prototype in
pmode.h

Return value
Current operating system mode. The return value will be one of the following values:

Return value	Meaning

PM_realMode	The application is running in 80x86 real mode.
PM_286	The application is running in 16 bit protected mode
PM_386	The application is running in 32 bit protected mode

PM_getVGAColorTextSelector

Returns a selector to the color VGA text video memory.

Syntax

```
uint PM_getVGAColorTextSelector(void);
```

Prototype in

pmode.h

Return value

Selector to the color VGA text video memory, or 0 on failure.

Description

Returns a selector to the color VGA text video memory, normally located at segment 0xB800 in real mode memory. This is a pre-built selector and will be re-used by all subsequent calls to this routine.

This function is also valid for 32 bit protected mode applications (including Win32), however you need to access the memory using a far pointer. Most 32 bit compilers do not support this, but you can use the PM_getByte/PM_setByte family of functions to access his memory if speed is not critical (otherwise you will have to write the code in assembler).

See also

PM_createSelector, PM_freeSelector, PM_getBIOSSelector, PM_getVGASelector, PM_getByte, PM_getWord, PM_getLong, PM_setByte, PM_setWord, PM_setLong, PM_memcpynf, PM_memcpyfn

PM_getVGAGraphSelector

Returns a selector to the color VGA graphics video memory.

Syntax

```
uint PM_getVGAGraphSelector(void);
```

Prototype in

pmode.h

Return value

Selector to the color VGA graphics video memory, or 0 on failure.

Description

Returns a selector to the color VGA text video memory, normally located at segment 0xA000 in real mode memory. This is a pre-built selector and will be re-used by all subsequent calls to this routine.

This function is also valid for 32 bit protected mode applications (including Win32), however you need to access the memory using a far pointer. Most 32 bit compilers do not support this, but you can use the PM_getByte/PM_setByte family of functions to access his memory if speed is not critical (otherwise you will have to write the code in assembler).

See also

PM_createSelector, PM_freeSelector, PM_getBIOSSelector, PM_getVGASelector, PM_getByte, PM_getWord, PM_getLong, PM_setByte, PM_setWord, PM_setLong, PM_memcpynf,PM_memcpyfn

PM_getVGAMonoTextSelector

Returns a selector to the monochrome text video memory.

Syntax

```
uint PM_getVGAMonoTextSelector(void);
```

Prototype in

pmode.h

Return value

Selector to the monochrome text video memory, or 0 on failure.

Description

Returns a selector to the monochrome text video memory, normally located at segment 0xB000 in real mode memory. This is a pre-built selector and will be re-used by all subsequent calls to this routine.

This function is also valid for 32 bit protected mode applications (including Win32), however you need to access the memory using a far pointer. Most 32 bit compilers do not support this, but you can use the PM_getByte/PM_setByte family of functions to access his memory if speed is not critical (otherwise you will have to write the code in assembler).

See also

PM_createSelector, PM_freeSelector, PM_getBIOSSelector, PM_getVGASelector, PM_getByte, PM_getWord, PM_getLong, PM_setByte, PM_setWord, PM_setLong, PM_memcpynf, PM_memcpyfn

PM_getVGASelector

Returns a selector to the VGA video memory for current video mode.

Syntax

```
uint PM_getVGASelector(void);
```

Prototype in

pmode.h

Return value
Selector to VGA video memory for current video mode.

Description
Returns a selector to the VGA video memory, depending on the current video mode. This will return a pre-built selector to the proper video memory location depending on whether the current video mode is color text mode, monochrome text mode or color graphics mode.

This function is also valid for 32 bit protected mode applications (including Win32), however you need to access the memory using a far pointer. Most 32 bit compilers do not support this, but you can use the PM_getByte/PM_setByte family of functions to access his memory if speed is not critical (otherwise you will have to write the code in assembler).

See also
PM_createSelector, PM_freeSelector, PM_getBIOSSelector, PM_getByte, PM_getWord, PM_getLong, PM_setByte, PM_setWord, PM_setLong, PM_memcpynf, PM_memcpyfn

PM_getWord

Reads a 16 bit word from a far address.

Syntax
```
ushort PM_getWord(uint s, uint o);
```

Prototype in
pmode.h

Parameters
s	Selector for word to read
o	Offset of word to read

Return value
Word read from the specified location.

Description
Reads a word from a far address. In 16 bit protected mode you can simply access the memory using a normal far pointer, however in 32 bit protected mode, most compilers do not directly support 48 bit far pointers (16 bit selector, 32 bit offset). This function provides the ability to read memory using a different selector to the default DS selector in 32 bit protected modes.

For speed critical code, you should write the code directly in 32 bit assembler.

See also
PM_getByte, PM_getLong, PM_setByte, PM_setWord, PM_setLong, PM_memcpynf, PM_memcpyfn

PM_int386

Issues a protected mode interrupt.

Syntax

```
int PM_int386(int intno,PMREGS *in,PMREGS *out);
```

Prototype in

pmode.h

Parameters

intno	Software interrupt number to generate
in	Value of 32 bit registers to load before interrupt
out	Place to store 32 bit register values after interrupt

Return value

Value of the EAX register.

Description

Issues a protected mode software interrupt, and supports the passing of full 32 bit register values to the interrupt routine. The value of the 32 bit general purpose registers stored in the *in* strcture are loaded before the interrupt is generated, and the return values for all the 32 bit general purpose registers is stored in the *out* structure after the interrupt has been processed. Both *in* and *out* may point to the same register structure if desired.

Note that this routine issues a *protected mode* interrupt so cannot be used for calling real mode interrupt handlers; you must use PM_int86 for that.

See also

PM_int386x, PM_int86, PM_int86x

PM_int386x

Issues a protected mode interrupt.

Syntax

```
int PM_int386x(int intno,PMREGS *in,PMREGS *out,PMSREGS
*sregs);
```

Prototype in

pmode.h

Parameters

intno	Software interrupt number to generate
in	Value of 32 bit registers to load before interrupt
out	Place to store 32 bit register values after interrupt
sregs	Pointer to structure holding the segment register values

Return value

Value of the EAX register.

Description

Issues a protected mode software interrupt, and supports the passing of full 32 bit general purpose and segment register values to the interrupt routine. The value of the 32 bit general purpose registers stored in the *in* structure and the segment registers values in *sregs* are loaded before the interrupt is generated. Upon return the values for all the 32 bit general purpose registers is stored in the *out* structure and the segment registers in the *sregs* structure. Both *in* and *out* may point to the same register structure if desired.

Note that this routine issues a *protected mode* interrupt so cannot be used for calling real mode interrupt handlers; you must use PM_int86 for that. Note also that the values in *sregs must* be valid protected mode selector values, so you must either load valid values or 0 into *all* members of the structure.

See also

PM_segread, PM_int386, PM_int86, PM_int86x

PM_int86

Issues a real mode interrupt.

Syntax

```
int PM_int86(int intno,RMREGS *in,RMREGS *out);
```

Prototype in

pmode.h

Parameters

intno	Software interrupt number to generate
in	Value of 16 bit real mode registers to load before interrupt
out	Place to store 16 bit real mode register values after interrupt

Return value

Value of the AX register.

Description

Issues a real mode software interrupt from protected mode code. The value of the 16 bit general purpose registers stored in the *in* strcture are loaded before the interrupt is generated, and the return values for all the 16 bit general purpose registers is stored in the *out* structure after the interrupt has been processed. Both *in* and *out* may point to the same register structure if desired.

Note that this routine issues a *real mode* interrupt, and does not do any translation on parameters passed to the real mode code. The real mode code will be expecting any addresses passed in to be proper real mode addresses for memory located in the first 1Mb of real mode memory. You will need to allocate a real mode transfer buffer using

PM_allocRealSeg and copy the information temporarily into this transfer buffer. You cannot simply pass normal protected mode addresses to the real mode code.

See also
PM_int86x, PM_allocRealSeg

PM_int86x

Issues a real mode interrupt.

Syntax
```
int PM_int86x(int intno,RMREGS *in,RMREGS *out,RMSREGS
*sregs);
```

Prototype in
pmode.h

Parameters

intno	Software interrupt number to generate
in	Value of 16 bit real mode registers to load before interrupt
out	Place to store 16 bit real mode register values after interrupt
sregs	Pointer to structure holding the segment register values

Return value
Value of the AX register.

Description
Issues a real mode software interrupt from protected mode. The value of the 16 bit general purpose registers stored in the *in* strcture and the segment registers values in *sregs* are loaded before the interrupt is generated. Upon return the values for all the 16 bit general purpose registers is stored in the *out* structure and the segment registers in the *sregs* structure. Both *in* and *out* may point to the same register structure if desired.

Note that this routine issues a *real mode* interrupt, and does not do any translation on parameters passed to the real mode code. The real mode code will be expecting any addresses passed in to be proper real mode addresses for memory located in the first 1Mb of real mode memory. You will need to allocate a real mode transfer buffer using PM_allocRealSeg and copy the information temporarily into this transfer buffer. You cannot simply pass normal protected mode addresses to the real mode code.

See also
PM_int86, PM_allocRealSeg

PM_mapPhysicalAddr

Maps a physical memory region to a 32 bit near pointer.

Syntax

```
void *PM_mapPhysicalAddr(ulong base,ulong limit);
```

Prototype in

pmode.h

Parameters

base	32 bit physical base address of memory to map
limit	32 bit limit for memory to map. Must be page aligned.

Return value

32 bit near pointer to mapped physical memory location.

Description

Maps a region of physical memory to a 32 bit near pointer that can be used to directly access the physical memory from 32 bit protected mode applications without the use of a separate selector. Normally physical memory addresses above 1Mb (such as a linear framebuffer for high performance SuperVGA graphics cards) cannot be directly accessed, and must first be mapped into the processes linear address space using this function. This function is only valid for 32 bit protected mode. To access physical memory from 16 bit protected mode, you must allocate a selector to the memory using PM_createSelector.

Note that this mapping cannot be freed, so you should attempt to re-use the same physical memory mapping for the duration of the program.

See also

PM_createSelector

PM_mapRealPointer

Maps a real memory address to a protected mode address.

Syntax

```
void PM_mapRealPointer(uint *sel,uint *off,uint r_seg,uint
r_off);
```

Prototype in

pmode.h

Parameters

sel	Place to store protected mode selector for mapped memory
off	Place to store protected mode offset for mapped memory
r_seg	Real mode segment to map
r_off	Real mode offset to map

Description

Maps a real mode memory address to a protected mode address. Real mode memory addresses cannot be used directly in protected mode, so this function is used to get a temporary protected mode pointer to a real mode memory address, so that you can directly access the memory from protected mode. For 32 bit protected mode, the memory must be accessed via a 48 bit far pointer, either using the PM_getByte style functions or directly from assembly language.

Note that in 16 bit protected mode the selector return by this routine will be re-used the next time the routine is called, so you should not remember the selector between calls to this routine.

See also

PM_allocRealSeg, PM_getByte, PM_getWord, PM_getLong

PM_memcpyfn

Copies a block of memory from a near address to a protected mode far address.

Syntax

```
void PM_memcpyfn(uint dst_s,uint dst_o,void *src,uint n);
```

Prototype in

pmode.h

Parameters

dst_s	Selector for destination memory block
dst_o	Offset for destination memory block
src	Near pointer to source block
n	Number of bytes to copy

Description

Copies a block from a near address to a protected mode far address. Normally you cannot directly access memory in a different segment from 32 bit protected mode code, as most 32 bit compilers do not directly support 48 bit far pointers (16 bit segment, 32 bit offset). This routine can be used for copying memory blocks between far memory and near memory (usually to copy data to and from real mode memory).

See also

PM_memcpynf, PM_getByte, PM_getWord, PM_getLong

PM_memcpynf

Copies a block of memory from a protected mode far address to a near address.

Syntax

```
void PM_memcpynf(void *dst,uint src_s,uint src_o,uint n);
```

Prototype in

pmode.h

Parameters

dst	Near pointer to destination memory block
src_s	Selector for source memory block
src_o	Offset for source memory block
n	Number of bytes to copy

Description

Copies a block from a protected mode far address to a near address. Normally you cannot directly access memory in a different segment from 32 bit protected mode code, as most 32 bit compilers do not directly support 48 bit far pointers (16 bit segment, 32 bit offset). This routine can be used for copying memory blocks between far memory and near memory (usually to copy data to and from real mode memory).

See also

PM_memcpyfn,

PM_getByte, PM_getWord, PM_getLong

PM_segread

Reads the value of all segment registers.

Syntax

```
void PM_segread(PMSREGS *sregs);
```

Prototype in

pmode.h

Parameters

sregs	Pointer to segment register block to fill

Description

Reads the value of all the 32 bit segment registers and stores the values in the *sregs* register block. If you use the PM_int386x routine to call 32 bit protected mode interrupt handlers (such as DPMI functions) you must load the segment registers with valid protected mode selectors. This routine can be used to get the current values of all these selectors before calling 32 bit interrupt handlers.

See also

PM_int386x

PM_setByte

Store a byte at a far address.

Syntax

```
void PM_setByte(uint s, uint o, uchar v);
```

Prototype in
pmode.h

Parameters
s	Selector of address to store value in
o	Offset of address to store value in
v	Value to store

Description
Stores a byte at a far address. In 16 bit protected mode you can simply access the memory using a normal far pointer, however in 32 bit protected mode, most compilers do not directly support 48 bit far pointers (16 bit selector, 32 bit offset). This function provides the ability to write memory using a different selector to the default DS selector in 32 bit protected modes.

For speed critical code, you should write the code directly in 32 bit assembler.

See also
PM_getByte,PM_getWord,PM_getLong,PM_setWord,PM_setLong,PM_memcpynf,PM_memcpyfn

PM_setLong

Store a long at a far address.

Syntax
```
void PM_setLong(uint s, uint o, ulong v);
```

Prototype in
pmode.h

Parameters
s	Selector of address to store value in
o	Offset of address to store value in
v	Value to store

Description
Stores a long at a far address. In 16 bit protected mode you can simply access the memory using a normal far pointer, however in 32 bit protected mode, most compilers do not directly support 48 bit far pointers (16 bit selector, 32 bit offset). This function provides the ability to write memory using a different selector to the default DS selector in 32 bit protected modes.

For speed critical code, you should write the code directly in 32 bit assembler.

See also
PM_getByte,PM_getWord,PM_getLong,PM_setByte,PM_setWord,PM_memcpynf,PM_memcpyfn

PM_setWord

Store a word at a far address.

Syntax

```
void PM_setWord(uint s, uint o, ushort v);
```

Prototype in

pmode.h

Parameters

s	Selector of address to store value in
o	Offset of address to store value in
v	Value to store

Description

Stores a word at a far address. In 16 bit protected mode you can simply access the memory using a normal far pointer, however in 32 bit protected mode, most compilers do not directly support 48 bit far pointers (16 bit selector, 32 bit offset). This function provides the ability to write memory using a different selector to the default DS selector in 32 bit protected modes.

For speed critical code, you should write the code directly in 32 bit assembler.

See also

PM_getByte,PM_getWord,PM_getLong,PM_setWord,PM_setLong,PM_memcpynf,PM_memcpyfn

SV_beginDirectAccess

Enables direct framebuffer access.

Syntax

```
void SV_beginDirectAccess(void);
```

Prototype in

svga.h

Description

Enables direct framebuffer access so that you can directly render to the banked or linear framebuffer memory. Note that calling this function is *absolutely* necessary when using hardware acceleration, as this function and the corresponding SV_endDirectAccess correctly arbitrate between the hardware accelerator graphics engine and your direct framebuffer writes.

See also

SV_endDirectAccess

SV_beginLine

Sets up for fast batched line drawing.

Syntax

```
void SV_beginLine(void);
```

Prototype in

svga.h

Description

Enables fast batched line drawing. This function sets up the hardware for the highest performance line drawing using the SV_lineFast function rather than the standard SV_line function. Batching lines together provides the absolute fastest way to draw multiple lines, however you can only call the line drawing functions between calls to SV_beginLine and SV_endLine.

See also

SV_lineFast, SV_endLine

SV_beginPixel

Sets up for fast batched pixel drawing.

Syntax

```
void SV_beginPixel(void);
```

Prototype in

svga.h

Description

This function sets up the hardware for the highest performance pixel drawing using the SV_putPixelFast function rather than the standard SV_putPixel function. Batching pixels together provides the absolute fastest way to draw multiple pixels, however you can only call the pixel drawing functions between calls to SV_beginPixel and SV_endPixel.

See also

SV_putPixelFast, SV_endPixel

SV_clear

Clears the currently active page in the framebuffer.

Syntax

```
void SV_clear(ulong color);
```

Prototype in

svga.h

Parameters

color	Color to clear the framebuffer with (framebuffer format)

Description
Clears the currently active display page to the specified color. The color value *must* be in the correct format for the current graphics mode (use SV_rgbColor to pack HiColor and TrueColor RGB colors into a framebuffer color.

SV_endDirectAccess

Disables direct framebuffer access.

Syntax
```
void SV_endDirectAccess(void);
```

Prototype in
svga.h

Description
Disables direct framebuffer access so that you can use the accelerator functions to draw the framebuffer memory. Note that calling this function is *absolutely* necessary when using hardware acceleration, as this function and the corresponding SV_endDirectAccess correctly arbitrate between the hardware accelerator graphics engine and your direct framebuffer writes.

See also
SV_beginDirectAccess

SV_endLine

Ends fast batched line drawing.

Syntax
```
void SV_endLine(void);
```

Prototype in
svga.h

Description
Ends fast batched line drawing. This function restores the hardware back to the default state after drawing batched lines with the SV_lineFast function. Batching lines together provides the absolute fastest way to draw multiple lines, however you can only call the line drawing functions between calls to SV_endLine and SV_endLine.

See also
SV_lineFast, SV_beginLine

SV_endPixel

Ends fast batched pixel drawing.

Syntax

```
void SV_endPixel(void);
```

Prototype in

svga.h

Description

Ends fast batched pixel drawing. This function restores the hardware back to the default state after drawing batched pixels with the SV_putPixelFast function. Batching pixels together provides the absolute fastest way to draw multiple pixels, however you can only call the pixel drawing functions between calls to SV_endPixel and SV_endPixel.

See also

SV_putPixelFast, SV_beginPixel

SV_getDefPalette

Returns a pointer to the default VGA palette values.

Syntax

```
SV_palette *SV_getDefPalette(void);
```

Prototype in

svga.h

Return value

Pointer to the default VGA palette values.

Description

This function returns a pointer to the default values normally programmed into the VGA palette, which are always programmed during the SV_setMode function of the SuperVGA Kit. Note tha the values are in 8 bits per primary format rather than the 6 bits per primary format normally used by the VGA palette.

SV_getModeInfo

Obtains information about a specific graphics mode.

Syntax

```
bool SV_getModeInfo(ushort mode, SV_modeInfo *modeInfo);
```

Prototype in

svga.h

Parameters

mode	Video mode to get information for
modeInfo	Pointer to buffer to return the mode information

Return value

True if the mode is valid, false if the mode is invalid.

Description

Returns the video mode information for the specified internal graphics mode number. The mode number must be valid, or this routine will return FALSE. Check the global modeList array which contains a list of available graphics modes that you can use. For the structure of the modeInfo block that is filled in, see the data structure reference at the end of this document.

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

```
#define svHaveDoubleBuffer  0x0001
#define svHaveVirtualScroll 0x0002
#define svHaveBankedBuffer  0x0004
#define svHaveLinearBuffer  0x0008
#define svHaveAccel2D       0x0010
#define svHaveDualBuffers   0x0020
#define svHaveHWCursor      0x0040
#define svHave8BitDAC       0x0080
#define svNonVGAMode        0x0100
```

The *svHaveDoubleBuffer* flag is used to determine whether the graphics mode can support hardware double buffering used for flicker free animation. If this bit is 0, then the application cannot start a double buffered video mode (usually because there is not enough display memory for two video buffers). Note that if the application is running on VBE 2.0 or lower rather than VBE/AF, some devices that can support double buffering with VBE/AF may not be able to support double buffering with VBE 2.0 or lower (Diamond Viper's and card based on the Weitek P9000 for example).

The *svHaveVirtualScroll* 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 (double buffering and virtual scrolling are separate, since some controllers may support one but not the other; most support both).

The *svHaveBankedBuffer* 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, so it is important that this bit is checked before blindly assuming that banked framebuffer access is always available. In this case where a banked framebuffer is not available, a linear framebuffer mode will be always provided.

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

The *svHaveAccel2D* 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. It is also zero for all VBE 2.0 and lower devices since they don't support acceleration.

The *svHaveDualBuffers* flag is used to determine if double buffering is implemented as two distinct framebuffer's, 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. Note that this flag is only relevant to VBE/AF hardware acceleration support.

The *svHaveHWCursor* 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 *svHave8BitDAC* 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. When running in VBE 2.0 modes, the 8 bit DAC mode will be selected if the use8BitDAC flag is set to true when the mode is initialized. However under VBE/AF the 8 bit DAC mode is not selectable and will always be used if available. Internally the SuperVGA Kit's palette setting functions automatically convert from 8 bits per primary format down to the 6 bits per primary format on controllers that cannot support 8 bit DAC modes.

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.

See also
SV_getModeName

SV_getModeName

Builds a string representing the capabilities of the graphics mode.

Syntax

```
ushort SV_getModeName(char *buf,SV_modeInfo *mi,ushort
mode,bool useLinear);
```

Prototype in

svga.h

Parameters

buf	Place to format the name of the graphics mode
mi	Graphics mode info buffer from SV_getModeInfo
mode	VBE mode number used
useLinear	True if linear mode should be selected if available

Return value

Mode number to use when starting the graphics mode

Description

This is a simple utility function that builds a formatted name for the graphics mode that details the modes capabilities, and also automatically adds the svLinearBuffer flag to the graphics mode to enable the linear framebuffer mode if the linear framebuffer is available in that mode. If the useLinear flag is set to false, the banked version of the mode will only ever be used.

See also

SV_getModeInfo

SV_init

Initializes the SuperVGA Kit and detects the underlying graphics hardware.

Syntax

```
SV_devCtx *SV_init(bool useVBEAF);
```

Prototype in

svga.h

Parameters

useVBEAF	True if VBE/AF should be used if available

Return value

Pointer to global device context block on success, or NULL of no hardware found.

Description

Detects if a VESA VBE or VBE/AF compliant graphics card is installed in the system, and initializes the graphics library if one is found. If suitable graphics hardware is not detected in the system, this function will return a value of NULL, otherwise it will return a pointer to the global device context block for the SuperVGA Kit. This is a pointer to the global variable structure used by the SuperVGA Kit so that you can directly access the SuperVGA Kit global variable even if the code resides in a different DLL. The version number of the VBE device detected is stored in the *VBEVersion* field of this structure when this function returns.

If a VBE/AF device is detected, the function will return a VBE version number of 0x200 (2.0) rather than the VBE/AF version number, and you can check the AFDC global device context pointer to get further information about the installed VBE/AF device. If VBE/AF is not found, the AFDC pointer will be set to NULL and the real VBE version number will be returned. For VBE 1.2 this is 0x102, for VBE 2.0 this is 0x200.

If the *useVBEAF* flag is set to true when this function is called, by default the initialization code will first search for a functioning VBE/AF device driver for the graphics card. If this is not found it then searches for a standard VBE 2.0 or lower device. If the *useVBEAF* flag is set to false, we skip the search for the VBE/AF driver and simply use the standard VBE driver if present.

SV_initRMBuf

Initializes the real mode transfer buffer for the library.

Syntax

```
void SV_initRMBuf(void);
```

Prototype in

svga.h

Description

This function initializes the real mode transfer buffer used by the SuperVGA Kit for communicating with VESA VBE devices and other real mode BIOS routines. You dont normally need to call this function directly, but if you need to make use of any of the functions in the VESAVBE.C module before calling the SV_init function you will need to ensure that this function is called first.

SV_line

Draws a solid line to the currently active display buffer in specified color.

Syntax

```
void SV_line(int x1,int y1,int x2,int y2,ulong color);
```

Prototype in

svga.h

Parameters

x1	First X coordinate of the line to draw
y1	First Y coordinate of the line to draw
x2	Second X coordinate of the line to draw
y2	Second Y coordinate of the line to draw
color	Color to draw the line in (framebuffer format)

Description

Draws a line from the point (x1,y1) to (x2,y2) in the specified color. The color value *must* be in the correct format for the current video mode (use SV_rgbColor to pack HiColor and TrueColor RGB values into the framebuffer format). The line is drawn on the currently active display page (which may possibly be hidden from view).

This function automatically uses the hardware accelerated line drawing capabilities of the VBE/AF device if accelerated line drawing is available, otherwise it simply calls the standard software line drawing routines in the SuperVGA Kit.

Note that the VBE/AF accelerated line drawing functions take pixel coordinates in 16.16 fixed point format and can draw lines with sub-pixel precision. This function only takes integer coordinates, so if you need sub-pixel precision line drawing you might want to call the VBE/AF line drawing code directly for better precision.

See also

SV_lineFast

SV_lineFast

Draws a batched solid line to the currently active display buffer in specified color.

Syntax

```
void SV_lineFast(int x1,int y1,int x2,int y2,ulong color);
```

Prototype in

svga.h

Parameters

x1	First X coordinate of the line to draw
y1	First Y coordinate of the line to draw
x2	Second X coordinate of the line to draw
y2	Second Y coordinate of the line to draw
color	Color to draw the line in (framebuffer format)

Description

This function is identical to the standard line drawing code, but it skips certain steps required to set up the hardware for fast line drawing. If you call this function you must call the SV_beginLine function before drawing multiple fast lines, and call SV_endLine when you are done. Note that between these two calls you cannot call any other graphics output functions.

See also

SV_beginLine, SV_endLine, SV_line

SV_putPixel

Plots a pixel at the specified location.

Syntax
```
void SV_putPixel(int x,int y,ulong color);
```

Prototype in
svga.h

Description
Plots a pixel at the specified (x,y) location in the specified color. The color value *must* be in the correct format for the current video mode (use SV_rgbColor to pack HiColor and TrueColor RGB values into the framebuffer format). The pixel is drawn on the currently active display page (which may possibly be hidden from view).

See also
SV_putPixelFast

SV_putPixelFast

Plots a batched pixel at the specified location.

Syntax
```
void SV_putPixelFast(int x,int y,ulong color);
```

Prototype in
svga.h

Description
This function is identical to the standard pixel drawing code, but it skips certain steps required to set up the hardware for fast pixel plotting. If you call this function you must call the SV_beginPixel function before drawing multiple fast pixels, and call SV_endPixel when you are done. Note that between these two calls you cannot call any other graphics output functions.

See also
SV_beginPixel, SV_endPixel, SV_putPixel

SV_restoreMode

Restore text mode operation after graphics mode has been used.

Syntax
```
void SV_restoreMode(void);
```

Prototype in
svga.h

Description
Restores the previous video mode active before the SV_setMode routine was called. Also correctly restores the VGA 50 line mode if it was previously active.

See also
SV_setMode, SV_setVirtualMode

SV_rgbColor

Builds a framebuffer pixel color given 8 bit RGB tuples.

Syntax
```
ulong SV_rgbColor(uchar r,uchar g,uchar b);
```

Prototype in
svga.h

Parameters
r	Red component for the color
g	Green component for the color
b	Blue component for the color

Return value
Packed framebuffer pixel value for the specified RGB color.

Description
Packs a set of 8 bit RGB tuples into a framebuffer pixel value for passing to the primitive drawing routines that is appropriate for the current graphics mode. This routine is intended to work with RGB video modes such as the 15, 16, 24 and 32 bits per pixel modes (in 8 bit modes it will packed it into a simple 3:3:2 style pixel, so you can set up your own pseudo RGB palette if you so desire).

Note that for maximum speed you may wish to convert this function into a macro in your own code, and hand code special case versions that pack the pixels directly for each supported graphics mode.

SV_setActivePage

Sets the currently active page that all output is drawn to.

Syntax
```
void SV_setActivePage(int page);
```

Prototype in
svga.h

Parameters
page	Page index to make active (0+)

Description

Sets the currently active video page for output. When the active page is changed, all output is sent to the new page which may be hidden. This is generally used to implement double buffering for smooth animation.

Note that you should check if the graphics mode supports hardware double buffering (svHaveDoubleBuffer flag for SV_modeInfo block) before you use this function.

See also

SV_setVisualPage

SV_setBank

Changes the currently active framebuffer bank.

Syntax

```
void SV_setBank(int bank);
```

Prototype in

svga.h

Parameters

bank	New 64Kb bank to make active

Description

'C' callable bank switch routine to set the current read/write bank to the specified value. Assembler functions don't call this routine but call the SV_setBankASM register level version in _SVGASDK.asm which is faster.

SV_setDisplayStart

Changes the display start address for virtual scrolling.

Syntax

```
void SV_setDisplayStart(int x,int y,bool waitVRT);
```

Prototype in

svga.h

Parameters

x	New display start X coordinate
y	New display start Y coordinate
waitVRT	True if we should wait for the vertical retrace

Description

Sets the CRTC display starting address to the specified value. You can use this routine to implement hardware virtual scrolling. If the *waitVRT* flag is false, the routine will not wait for a vertical retrace before programming the CRTC starting address, otherwise the routine will sync to a vertical retrace. Under VBE 1.2 it is not guaranteed what the behavior will be (some wait and some don't).

Note that if the controller is in a NonVGA mode, you should *not* attempt to do any vertical retrace synching in your own code. In NonVGA modes the VGA registers do not exist, and your code will most likely hang the machine waiting for a VGA retrace that never occurs.

Note that you should check if the graphics mode supports hardware virtual scrolling (svHaveVirtualScroll flag for SV_modeInfo block) before you use this function.

SV_setMode

Initialize a standard graphics mode.

Syntax

```
bool SV_setMode(ushort mode,bool use8BitDAC,bool
useVirtualBuffer,int numBuffers);
```

Prototype in

svga.h

Parameters

mode	Graphics mode to initialize (with flags)
use8BitDAC	True If 8 bit DAC should be used if available
useVirtualBuffer	True to use virtual linear buffer if available
numBuffers	Number of display buffers to allocate if multi-buffering

Return value

True if the mode was initialized, false on error.

Description

Set the specified video mode, given the mode number. *Do not* pass old style hard coded VBE mode numbers to this routine (i.e.: 0x101 for 640x480x256). Although these *may* still work, the VBE 2.0 and VBE/AF method is to search through the list of available graphics modes for the one that has the desired resolution and color depth. This will allow your code to work with all custom resolutions provided by different OEM VBE drivers (like out UniVBE driver). Have a look at the code in the HELLOVBE.C file that demonstrates how to start any video mode given a user specified resolution.

The mode number that you pass in can have a number of flags logically 'or' with the standard mode number:

```
#define svDontClear     0x8000
#define svLinearBuffer  0x4000
#define svMultiBuffer   0x2000
```

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

The *svLinearBuffer* flag is used to enable the hardware linear framebuffer version of the graphics mode. On many controllers, the banked and linear framebuffer's 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 *svHaveLinearBuffer* flag in the mode attributes field to determine if this is supported in the selected graphics mode. Note that this flag is *not* used to initialize a virtual linear framebuffer mode.

The *svMultiBuffer* flag must be set if the application intends to use multi buffering in the graphics mode for VBE/AF devices. When the graphics 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 *svHaveMultiBuffer* flag in the mode attributes field to determine if the selected graphics mode supports multi buffering. You must also pass in the number of buffers that you wish to use in the *numBuffers* parameter, so that VBE/AF devices can set aside the correct amount of video memory for display buffer's and for offscreen memory for storing bitmaps. If multi buffering is active, there will be less memory available for storing offscreen bitmaps in video memory. Under VBE 2.0 and lower devices this flag is simply ignored.

If the *'useVirtualBuffer'* flag is set to true and a virtual linear framebuffer is available, the mode will be initialized as a virtual linear mode rather than a banked framebuffer mode.

See also
SV_setVirtualMode

SV_setPalette

Builds a framebuffer pixel color given 8 bit RGB tuples.

Syntax
```
void SV_setPalette(int start,int num,SV_palette *pal,int
maxProg,int numBuffers);
```

Prototype in
svga.h

Parameters

start	Starting hardware index to begin programming at
num	Number of palette entries to program
pal	Pointer to array of values to program
maxProg	Maximum number of values to program during vertical retrace

Description

This function set the specified palette entries, by either directly programming the VGA hardware (for VBE 1.2 and below) or by calling the VBE 2.0 palette setting routines. This routine avoids 'snow' effects on older systems by only programming *maxProg* values per vertical retrace interval. If you set *maxProg* to 256, all values will be programmed at once and the palette set will be synched to a vertical retrace. If you set *maxProg* to -1, all values will be set at once and the routine will *not* wait for a vertical retrace before setting the values. For systems that cause snow, a good value of *maxProg* is about 100-120.

This routine is *fast* and will provide the fastest method of programming the palette that will work in all systems. Because of the way that palette values are programmed, color values will not be dropped on systems that have slower IO response, so you should *always* use this routine rather than programming the palette yourself. If the controller is in a NonVGA mode and you program the palette via the VGA registers (as opposed to having the VBE 2.0 or VBE/AF code do it) your code will most likely product no result and may well hang the machine waiting for a VGA retrace that never occurs.

Note that the buffer is expected to be in the correct format, which will be an array of SV_palette structures. This routine expects the palette values to be in 8 bits per primary format (not the usual VGA format of 6 bits per primary) and will do conversion on the fly between the 8 bits per primary format and 6 bit format if the hardware does not support an 8 bit wide DAC.

SV_setVirtualMode

Initialize a hardware virtual scrolling graphics mode.

Syntax

```
bool SV_setVirtualMode(ushort mode,int virtualX,int
virtualY,bool use8BitDAC,bool useVirtualBuffer);
```

Prototype in

svga.h

Parameters

mode	Graphics mode to initialize (with flags)
virtualX	Virtual width of desired for mode
virtualY	Virtual height of desired mode
use8BitDAC	True if 8 bit DAC should be used if available
useVirtualBuffer	True to use virtual linear buffer if available

Return value

True if the mode was initialized, false on error.

Description

Set the specified video mode, given the mode number. This routine is similar to the standard mode set routine, but it initializes a hardware virtual scrolling graphics mode of the specified dimensions. You can also pass the *svDoubleBuffer* flag to this routine if the hardware supports both virtual scrolling and double buffering to start a double buffered and virtual scrolling version of the mode (not all hardware supports this).

Once the mode has been initialized with a specified virtual size, you cannot change this size without resetting the graphics mode.

See also

SV_setMode

SV_setVisualPage

Sets the currently visible page that is currently being displayed.

Syntax

```
void SV_setVisualPage(int page,bool waitVRT);
```

Prototype in

svga.h

Parameters

page	Page index to make visible (0+)
waitVRT	True to wait for the vertical retrace

Description

Sets the currently visible video page. This is generally used to implement double buffering for smooth animation. If the *waitVRT* flag is false, the routine will not wait for a vertical retrace before programming the CRTC starting address, otherwise the routine will sync to a vertical retrace. Under VBE 1.2 it is not guaranteed what the behavior will be (some wait and some don't).

Note that if the controller is in a NonVGA mode, you should *not* attempt to do any vertical retrace synching in your own code. In NonVGA modes the VGA registers do not exist, and your code will most likely hang the machine waiting for a VGA retrace that never occurs.

Note that you should check if the graphics mode supports hardware double buffering (svHaveDoubleBuffer flag for SV_modeInfo block) before you use this function.

See also

SV_setActivePage

SV_writeText

Writes a string to the display in the 8x16 graphics font.

Syntax

```
void SV_writeText(int x,int y,char *str,ulong color);
```

Prototype in
svga.h

Parameters
x	Left coordinate of first character drawn
y	Top coordinate of first character drawn
str	String to draw to the display
color	Color to draw the string in

Description
Writes the text sting at the location x,y in the standard 8x16 VGA font, and the specified color. The background between the text is not erased. Note that the font itself is not read from the VGA but is stored in the library, and you can replace this with any font you like.

See also
SV_putPixelFast

ULZElapsedTime

Compute the elpased time between to timer counts.

Syntax
```
ulong ULZElpasedTime(ulong start,ulong finish);
```

Prototype in
ztimer.h

Parameters
start	Starting time for elpased count
finish	Ending time for elapsed count

Description
Returns the elpased time for the Ultra Long Period Zen Timer in units of the timers resolution (1/18th of a second under DOS). This function correctly computes the difference even if a midnight boundary has been crossed during the timing period.

See also
ULZReadTime

ULZReadTime

Reads the current time from the Ultra Long Period Zen Timer.

Syntax
```
ulong ULZReadTime(void);
```

Prototype in
ztimer.h

Return value
Current timer value in resolution counts.

Description
Reads the current Ultra Long Period Zen Timer and returns it's current count. You can use the ULZElapsedTime. function to find the elapsed time between two timer count reading.

See also
ULZElapsedTime

ULZTimerCount

Returns the current count for the Ultra Long Period Zen Timer.

Syntax
```
ulong ULZTimerCount(void);
```

Prototype in
ztimer.h

Return value
Count that has elapsed in resolution counts.

Description
Returns the current count hat has elapsed between calls to ULZTimerOn and ULZTimerOff in resolution counts.

See also
ULZTimerOn, ULZTimerOff

ULZTimerLap

Returns the current count for the Ultra Long Period Zen Timer and keeps it running.

Syntax
```
ulong ULZTimerLap(void);
```

Prototype in
ztimer.h

Return value
Count that has elapsed in resolution counts.

Description
Returns the current count hat has elapsed since the last call to ULZTimerOn in resolution counts. The time continues to run after this function is called so you can call this function repeatedly.

See also
ULZTimerOn, ULZTimerOff

ULZTimerOff

Stops the Ultra Long Period Zen Timer counting.

Syntax
```
void ULZTimerOff(void);
```

Prototype in
ztimer.h

Description
Stops the Ultra Long Period Zen Timer counting and latches the count. Once you have stopped the timer you can read the count with ULZTimerCount.

See also
ULZTimerOn, ULZTimerCount

ULZTimerOn

Starts the Ultra Long Period Zen Timer counting.

Syntax
```
void ULZTimerOn(void);
```

Prototype in
ztimer.h

Description
Starts the Ultra Long Period Zen Timer counting. Once you have started the timer, you can stop it with ULZTimerOff or you can latch the current count with ULZTimerLap.

See also
ULZTimerOff, ULZTimerLap

ULZTimerResolution

Returns the resolution of the Ultra Long Period Zen Timer.

Syntax

```
float ULZTimerResolution(void);
```

Prototype in

ztimer.h

Return value

Resolution of the time in seconds per timer count.

Description

Returns the resolution of the Ultra Long Period Zen Timer as a floating point value measured in seconds per timer count.

VF_available

Returns true if the 32 bit virtual framebuffer device is available.

Syntax

```
bool VF_available(void)
```

Prototype in

pmpro.h

Return value

True if virtual framebuffer device available, false if not.

Description

This routine determines if the virtual framebuffer device is available for providing a virtual, 32 bit linear framebuffer for SuperVGA graphics devices that do not have a hardware linear framebuffer. Under Windows 3.1 the virtual framebuffer device is DVA.386 (provided with Video for Windows and distributed as part of the WinDirect package) and under Windows '95 it is the standard VFLATD.386 virtual device driver. Under MSDOS, this currently is supported only is running under the DOS4GW DOS extender, and this function will fail when running under a DPMI environment such as a Windows or OS/2 DOS box.

This function may fail if an older version of DVA.386 is installed under Windows 3.1 or if the VFLATD.386 device is not installed under Windows '95. Older versions of DVA.386 did not included support for a 32 bit near virtual framebuffer.

See also

VF_init, VF_exit

VF_exit

Removes the virtual framebuffer handler.

Syntax

```
void VF_exit(void)
```

Prototype in

pmpro.h

Description

Removes the virtual framebuffer handling routines. This function *must* be called before the application terminates to ensure that the virtual framebuffer device is correctly de-initialised.

See also

VF_init,VF_available

VF_init

Installs the virtual framebuffer handler.

Syntax

```
void *VF_init(ulong baseAddr,int bankSize,int codeLen,void
*bankFunc)
```

Prototype in

pmpro.h

Parameters

baseAddr	Physical base address of framebuffer window
bankSize	Size of framebuffer window in Kb (either 4Kb or 64Kb)
codeLen	Length of the 32 bit protected mode bank switch function
bankFunc	Pointer to the 32 bit protected mode bank switch function

Return value

Near pointer to the memory mapped by the virtual device, or NULL on failure.

Description

Installs the virtual framebuffer device handler, for providing a virtual, 32 bit linear framebuffer for SuperVGA graphics devices that do not have a hardware linear framebuffer. In order to virtualise the framebuffer for the video card, you must provide a 32 bit relocateable bank switch function. The function will not be able to access any global memory, and must *not* contain a return statement at the end. The virtual framebuffer device can virtualise devices with a bank size of either 4Kb or 64Kb.

Once the framebuffer has been virtualised, the virtual framebuffer device will use the 386 page faulting mechanism to automatically map the proper region of the SuperVGA video memory into the virtualised framebuffer area. Note that you must be very careful with non-aligned memory accesses to the virtual framebuffer region. If you access the memory on a non-aligned region that lies across a page fault boundary, you will cause an infinite page fault loop, causing your application and the entire system to hang. Hence when accessing the virtualised framebuffer memory, you should always ensure that you never access the memory using unaligned accesses.

You should first call VF_available to determine if the device is available, but this function will also fail if the virtual device driver is not present and functioning.

See also
VF_exit,VF_available

WD_asciiCode

Macro to extract the ASCII code from the message field of the event structure.

Syntax
```
uchar WD_asciiCode(ulong message);
```

Prototype in
event.h

Parameters
message	Message to extract ASCII code from

Return value
ASCII code extracted from the message.

See also
WD_scanCode

WD_flushEvent

Flushes all events of a specified type from the event queue.

Syntax
```
void WD_flushEvent(uint mask);
```

Prototype in
event.h

Parameters
mask	Mask specifying the types of events that should be removed

Description
Flushes (removes) all pending events of the specified type from the event queue. You may combine the masks for different event types with a simple logical OR.

See also
WD_getEvent, WD_haltEvent, WD_peekEvent

WD_getEvent

Retrieves the next pending event from the event queue.

Syntax

```
bool WD_getEvent(WD_event *evt,uint mask);
```

Prototype in

event.h

Return value

True if an event was pending, false if not.

Parameters

evt	Pointer to structure to return the event info in
mask	Mask specifying the types of events that should be removed

Description

Retrieves the next pending event from the event queue, and stores it in a WD_event structure. The mask parameter is used to specifiy the type of events to be removed, and can be any logical combination of any of the following flags:

Flag	Extract event type
EVT_KEYDOWN	Key press events
EVT_KEYREPEAT	Key repeat events
EVT_KEYUP	Key release events
EVT_KEYEVT	Any keyboard event
EVT_MOUSEDOWN	Mouse down events
EVT_MOUSEUP	Mouse up events
EVT_MOUSEMOVE	Mouse movement events
EVT_MOUSEEVT	Any mouse event
EVT_MOUSECLICK	Any mouse up or down event
EVT_TIMERTICK	Timer tick events
EVT_EVERYEVT	All events

The *what* field of the event contains the event code of the event that was extracted. All application specific events should begin with the EVT_USEREVT code and build from there. Since the event code is stored in an integer, there is a maximum of 16 different event codes that can be distinguished (32 for the 32 bit version). You can store extra information about the event in the *message* field to distinguish between events of the same class (for instance the button used in a EVT_MOUSEDOWN event).

If an event of the specified type was not in the event queue, the *what* field of the event will be set to NULLEVT, and the return value will return false.

The EVT_TIMERTICK event is used to report that a specified time interval has elapsed since the last EVT_TIMERTICK event occurred. See WD_setTimerTick() for information on how to enable timer tick events and to set the timer interval.

See also
WD_flushEvent, WD_haltEvent, WD_peekEvent, WD_setTimerTick

WD_getMousePos

Get the current location of the mouse cursor.

Syntax
```
void WD_getMousePos(int *x,int *y);
```

Prototype in
event.h

Parameters

x	Pointer to place to store mouse cursor X coordinate
y	Pointer to place to store mouse cursor Y coordinate

Description
Returns the current coordinates of the mouse cursor in the x and y parameters. Normally the mouse coordinates are determined from the mouse movement events, but this function is provided in case you need to obtain the current mouse coordinates. Note that the coordinate are returned in screen coordinates scaled to the resolution that was specified when WD_startFullScreeen was called.

WD_haltEvent

Pauses until a specified event occurs.

Syntax
```
void WD_haltEvent(WD_event *evt,uint mask);
```

Prototype in
event.h

Parameters

evt	Pointer to structure to return the event info in
mask	Mask specifying the types of events that should be removed

Description
Halts program execution until an event of the type specified by mask has occurred, and is returned in the WD_event structure. The event that was caught is removed from the event queue and returned. The mask parameter is used to specifiy the type of events to be removed, and can be any logical combination of any of the following flags:

Flag	Extract event type
EVT_KEYDOWN	Key press events
EVT_KEYREPEAT	Key repeat events
EVT_KEYUP	Key release events
EVT_KEYEVT	Any keyboard event
EVT_MOUSEDOWN	Mouse down events
EVT_MOUSEUP	Mouse up events
EVT_MOUSEMOVE	Mouse movement events
EVT_MOUSEEVT	Any mouse event
EVT_MOUSECLICK	Any mouse up or down event
EVT_TIMERTICK	Timer tick events
EVT_EVERYEVT	All events

Note that WD_haltEvent will return immediately if there is already a pending event of the specified type waiting in the event queue. You can call WD_flushEvent to flush all pending events of the specified type to unconditionally halt program execution.

Note that it is not usually a good idea to ignore keyboard and mouse events with this function, as they can quickly fill up the event queue (especially mouse movement events!).

See also

WD_flushEvent, WD_getEventWD_peekEvent

WD_peekEvent

Peeks at the next pending event in the event queue.

Syntax

```
bool WD_peekEvent(WD_event *evt,uint mask);
```

Prototype in

event.h

Return value

True if an event is pending, false if not.

Parameters

evt	Pointer to structure to return the event info in
mask	Mask specifying the types of events that should be removed

Description

Peeks at the next pending event of the specified type in the event queue. The mask parameter is used to specifiy the type of events to be peeked at, and can be any logical combination of any of the following flags:

Flag	Extract event type
EVT_KEYDOWN	Key press events
EVT_KEYREPEAT	Key repeat events
EVT_KEYUP	Key release events
EVT_KEYEVT	Any keyboard event
EVT_MOUSEDOWN	Mouse down events
EVT_MOUSEUP	Mouse up events
EVT_MOUSEMOVE	Mouse movement events
EVT_MOUSEEVT	Any mouse event
EVT_MOUSECLICK	Any mouse up or down event
EVT_TIMERTICK	Timer tick events
EVT_EVERYEVT	All events

In contrast to WD_getEvent, the event is *not* removed from the event queue. You may combine the masks for different event types with a simple logical OR.

Refer to WD_getEvent for a list of event codes and a description of the WD_event structure.

See also

WD_flushEvent, WD_getEvent, WD_haltEvent

WD_postEvent

Posts a user defined event to the event queue

Syntax

```
bool WD_postEvent(uint what,ulong message,ulong modifiers);
```

Prototype in

event.h

Return value

True if event was posted, false if event queue is full.

Parameters

what	Type code for message to post
message	Event specific message to post
modifiers	Event specific modifier flags to post

Description

This routine is used to post user defined events to the event queue.

See also

WD_flushEvent, WD_getEvent, WD_peekEvent, WD_haltEvent

WD_restoreGDI

Restores normal GDI operation after running in full screen mode.

Syntax

```
void WD_restoreGDI(void);
```

Prototype in

event.h

Description

This function restores normal GDI operation after running in a full screen mode, and will ensure that the normal GDI desktop is restore to its original state. After WD_restoreGDI is called, the Window returned by WD_startFullScreen will have been destroyed and must not be used anymore.

See also

WD_startFullScreen

WD_repeatCount

Macro to extract the key repeat count from the message field of the event structure.

Syntax

```
short EVT_repeatCount(ulong message);
```

Prototype in

event.h

Parameters

message	Message to extract repeat count from

Return value

Repeat count extracted from the message.

See also

WD_asciiCode

WD_scanCode

Macro to extract the keyboard scan code from the message field of the event structure.

Syntax

```
uchar WD_scanCode(ulong message);
```

Prototype in

event.h

Parameters

message	Message to extract scan code code from

Return value

Keyboard scan code extracted from the message.

See also

WD_asciiCode

WD_setMouseCallback

Registers the mouse movement callback function.

Syntax

```
void WD_setMouseCallback(void (*moveCursor)(int x,int y));
```

Prototype in

event.h

Parameters

moveCursor	User supplied function called when mouse cursor is moved

Description

This function installs a user supplied mouse callback handler. The mouse callback handler gets called when the mouse cursor has changed position, and should be used by the application to draw the mouse cursor at the new location. WinDirect does not provide direct support for drawing the mouse cursor on the screen, but by registering your own callback with this function, you can easily draw your own mouse cursor at the correct location on the screen.

WD_setMousePos

Set current mouse cursor location.

Syntax

```
void WD_setMousePos(int x,int y);
```

Prototype in

event.h

Parameters

x	New mouse cursor X coordinate
y	New mouse cursor Y coordinate

Description

This function sets the current mouse cursor location to the specified x and y coordinates. Note that the coordinate are passed in screen coordinates.

WD_setSuspendAppCallback

Registers the suspend application callback function.

Syntax

```
void WD_setSuspendAppCallback(int (*saveState)(int flags));
```

Prototype in

event.h

Parameters

saveState User supplied function called when application needs to be suspended

Description

This function is used to register an application specific suspend function. This callback will be called when the user has pressed one of the standard Windows *System Keys*, such as Alt-Tab and Ctrl-Esc. When this happens, WinDirect needs to restore normal GDI operation to allow the user to switch away from the WinDirect application and back to normal fullscreen mode. The callback may elect to ignore suspend requests except at certain locations in the program, such as in the normal *pause* or *options* screens for realtime applications like games.

When WinDirect detects the need to switch back to GDI mode, it will call the registered callback with a combination of the following flags:

Flag	Meaning
WD_DEACTIVATE	WD_DEACTIVATE will be sent when the WinDirect needs to restore GDI operation. It is the reponsibility of the callback function to save the current state of the application so that it can be properly restored at a later date (such as the current screen information, game state etc). If the callback returns *true*, it is assumed that the state has been correctly saved and that WinDirect may now restore normal GDI operation and switch back to the desktop.
WD_REACTIVATE	WD_REACTIVATE will be sent when the user re-activates your WinDirect application again, indicating that the fullscreen mode has now been restored, and the application must re-initialise the application specific video mode (WinDirect will *not* restore the VGA or SuperVGA video mode that was active; your application must do this) and restore the current state of the application ready to begin normal processing again. The return code of the callback is ignore.

The normal default handler always returns false, and hence the default case is for WinDirect to ignore the system keys and not allow the user to switch back to the the normal GDI desktop.

WD_setTimerTick

Set the interval between WD_TIMERTICK events

Syntax

```
int WD_setTimerTick(int ticks);
```

Prototype in

event.h

Parameters

ticks	New value for timer tick interval (in milliseconds)

Return value

Old value of timer tick interval

Description

This routine sets the number of ticks between each posting of the WD_TIMERTICK event to the event queue. The WD_TIMERTICK event is off by default. You can turn off the posting of WD_TIMERTICK events by setting the tick interval to 0. Under Windows, one tick is approximately equal to 1/1000 of a second (millisecond).

WD_startFullScreen

Starts full screen WinDirect mode

Syntax

```
HWND WD_startFullScreen(HWND hwndMain,int x,int y);
```

Prototype in

event.h

Return value

Handle of fullscreen GDI window used for event capture, NULL on failure.

Parameters

hwndMain	Handle to main application window or NULL if no main window
x	X resolution to return mouse coordinates in
y	Y resolution to return mouse coordinates in

Description

This function attempts to put the systen into full screen VGA mode, thereby shutting down the GDI and giving the application full control of the hardware. This routine also puts up a full screen GDI window covering the entire desktop while in full screen mode, which is used for all subsequent message processing. Once in fullscreen mode, you should use the WinDirect event handling functions for interfacing with the mouse and keyboard. You may however subclass the GDI window handle returned and perform all your own message processing.

If your application also has a normal GDI main window, then you must pass a valid window handle in *hwndMain*, which will be used as the parent window for the fullscreen WinDirect window. WinDirect needs this window handle to be able to properly minimise the application when processing user requests to switch back to the normal GDI desktop. If however your application only runs in fullscreen mode, then you can simply pass a NULL for this parameter.

The *x* and *y* parameters are used by WinDirect to correctly scale the mouse coordinates when in full screen mode. If you intend to start a 320x200 video mode, you should pass the values of 320 and 200 to ensure mouse coordinates are properly scaled. Note that if you process window messages using the returned HWND, the mouse coordinates will be returned in the original GDI resolution and will have to be scaled accordingly.

See also
WD_restoreGDI

ZTimerInit

Initializes the Zen Timer Library for use.

Syntax
```
void ZTimerInit(void);
```

Prototype in
ztimer.h

Description
This function initializes the Zen Timer library, and *must* be called before any of the remaining Zen Timer library functions are called. If this function is not called, the library may still work but you will most likely get very strange results from the timers.

Data Structure Reference

The following is a detailed reference of all data structures and type definitions used by the SuperVGA Kit and associated libraries.

PMREGS

Structure describing the 32 bit protected mode general purpose register values. This structure is used to pass register information to and from the PM_int386() functions.

```
struct _PMDWORDREGS {
    ulong   eax,ebx,ecx,edx,esi,edi,cflag;
    };
struct _PMWORDREGS {
    ushort  ax,ax_hi;
    ushort  bx,bx_hi;
    ushort  cx,cx_hi;
    ushort  dx,dx_hi;
    ushort  si,si_hi;
    ushort  di,di_hi;
    ushort  cflag,cflag_hi;
    };
struct _PMBYTEREGS {
    uchar   al, ah; ushort:16;
    uchar   bl, bh; ushort:16;
    uchar   cl, ch; ushort:16;
    uchar   dl, dh; ushort:16;
    };
typedef union {
    struct  _PMDWORDREGS e;
    struct  _PMWORDREGS  x;
    struct  _PMBYTEREGS  h;
    } PMREGS;
```

Declaration in
pmode.h

PMSREGS

Structure describing the 16 or 32 bit protected mode selector values. This structure is used to pass register information to and from the PM_int386x() functions. Note that the values stored in here *must* be valid protected mode selectors. A selector of 0 is a valid value, so for registers that are not defined to have any specific value you must pre-load them with zeros (or alternatively load the current values with PM_segread).

```
typedef struct {
    ushort  es;
    ushort  cs;
    ushort  ss;
```

```
    ushort  ds;
    ushort  fs;
    ushort  gs;
    } PMSREGS;
```

Declaration in
pmode.h

RMREGS

Structure describing the 16 bit real mode general purpose register values. This structure is used to pass register information to and from the PM_callRealMode() and PM_int86() functions.

```
struct _RMWORDREGS {
    ushort ax, bx, cx, dx, si, di, cflag;
    };
struct _RMBYTEREGS {
    uchar   al, ah, bl, bh, cl, ch, dl, dh;
    };
typedef union {
    struct  _RMWORDREGS x;
    struct  _RMBYTEREGS h;
    } RMREGS;
```

Declaration in
pmode.h

RMSREGS

Structure describing the 16 bit real mode segment register values. This structure is used to pass register information to and from the PM_callRealMode() and PM_int86x() functions.

```
typedef struct {
    ushort  es;
    ushort  cs;
    ushort  ss;
    ushort  ds;
    } RMSREGS;
```

Declaration in
pmode.h

SV_devCtx

Structure describing the global device context block for the SuperVGA Kit. This structure encapsulates all the global variables that the SuperVGA Kit uses internally, and a pointer to this structure is returned to the calling application when the SuperVGA Kit is initialized.

```
typedef struct {
    int      VBEVersion;
    int      maxx;
    int      maxy;
    ulong    maxcolor;
    ulong    defcolor;
    int      maxpage;
    ulong    bytesperline;
    int      bitsperpixel;
    int      bytesperpixel;
    int      memory;
    long     linearAddr;
    ushort   *modeList;
    char     *OEMString;
    int      curBank;
    bool     haveVirtualBuffer;
    bool     haveDoubleBuffer;
    bool     haveVirtualScroll;
    bool     haveWideDAC;
    bool     haveAccel2D;
    bool     haveHWCursor;
    bool     virtualBuffer;
    void     *videoMem;
    ulong    originOffset;
    ushort   bankOffset;
    uchar    redMask;
    uchar    greenMask;
    uchar    blueMask;
    int      redPos;
    int      redAdjust;
    int      greenPos;
    int      greenAdjust;
    int      bluePos;
    int      blueAdjust;
    AF_devCtx *AFDC;
    } SV_devCtx;
```

Declaration in

svga.h

Field	Description
VBEVersion	VBE version detected. For VBE/AF, we return a value of 0x200 rather than the VBE/AF version number (check the AFDC structure for the VBE/AF version number).
maxx	Maximum framebuffer X coordinate
maxy	Maximum framebuffer Y coordinate
maxcolor	Maximum color value
defcolor	Default color value (White)
maxpage	Maximum framebuffer page index (multi-buffering)
bytesperline	Bytes per logical scanline for framebuffer access
bitsperpixel	Color depth for mode in bits per pixel
bytesperpixel	Number of bytes required to store a single pixel
memory	Memory on board in kilobytes

linearAddr	Physical address of linear framebuffer (do not write to!!)
modeList	Global list of all available graphics modes
OEMString	VBE OEM string read from controller
curBank	Current read/write bank for banked framebuffer access
haveVirtualBuffer	True if virtual linear framebuffering is available
haveMultiBuffer	True if multi buffering is available in selected modes
haveVirtualScroll	True if virtual scrolling is available in selected modes
haveWideDAC	True if 8 bit wide DAC is available
haveAccel2D	True if 2D acceleration is available in selected modes
haveHWCursor	True if hardware cursor is available in selected modes
virtualBuffer	True if currently using virtual linear framebuffer
videoMem	Pointer to start of framebuffer memory (32 bit near pointer)
originOffset	Offset to start of currently active framebuffer page. This is actually a 32 bit pointer to the start of the active page in 32 bit linear framebuffer modes, otherwise it is an offset with the current memory aperture.
bankOffset	Bank offset to start of currently active framebuffer page
redMask	Red color channel for packing RGB pixels
greenMask	Green color channel for packing RGB pixels
blueMask	Blue color channel for packing RGB pixels
redPos	Red color channel position for packing RGB pixels
redAdjust	Red color channel shift adjustment for packing RGB pixels
greenPos	Green color channel position for packing RGB pixels
greenAdjust	Green color channel shift adjustment for packing RGB pixels
bluePos	Blue color channel position for packing RGB pixels
blueAdjust	Blue color channel shift adjustment for packing RGB pixels
AFDC	Pointer to VBE/AF device context block. This pointer will be NULL if VBE/AF is not available, otherwise it will point to a valid VBE/AF device driver. You must pass this pointer directly to any of the VBE/AF accelerator functions if you call them directly.

SV_modeInfo

Structure describing the information about a particular graphics mode. Note that this mode information block is a superset of the VBE 2.0 mode info block and the VBE/AF mode info blocks. Internally the SuperVGA Kit convert's between the VBE 2.0 and VBE/AF internal formats into the generic format supported by the SuperVGA Kit, which allows you to write a single set of code that will transparently use either VBE 2.0 or VBE/AF.

```
typedef struct {
    ushort  Mode;
    ushort  Attributes;
    ushort  XResolution;
    ushort  YResolution;
    ushort  BytesPerScanLine;
    ushort  BitsPerPixel;
    ushort  NumberOfPages;
    uchar   RedMaskSize;
    uchar   RedFieldPosition;
```

```
uchar   GreenMaskSize;
uchar   GreenFieldPosition;
uchar   BlueMaskSize;
uchar   BlueFieldPosition;
uchar   RsvdMaskSize;
uchar   RsvdFieldPosition;
} SV_modeInfo;
```

Declaration in

svga.h

Field	Description
Mode	VBE Mode number used to fill in this mode info block.
Attributes	Set of logical flags representing the attributes of this particular graphics mode. It will be a logical combination of the following: **Value** — **Meaning** svHaveMultiBuffer — Mode supports multi buffering svHaveVirtualScroll — Mode supports virtual scrolling svHaveBankedBuffer — Mode supports banked framebuffer access svHaveLinearBuffer — Mode supports linear framebuffer access svHaveAccel2D — Mode supports 2D acceleration svHaveDualBuffers — Mode uses dual buffers for double buffering svHaveHWCursor — Mode supports a hardware cursor svHave8BitDAC — Mode uses an 8 bit palette DAC svNonVGAMode — Mode is a NonVGA mode svIsVBEMode — Flags that info is for a VBE, not VBE/AF.
XResolution	Horizontal resolution for the mode in pixels
YResolution	Vertical resolution for the mode in pixels
BytesPerScanLine	Bytes per horizontal scanline for framebuffer access
BitsPerPixel	Color depth for mode in bits per pixel
NumberOfPages	Number of display pages supported by the mode
RedMaskSize	Size of direct color red channel mask
RedFieldPosition	Bit position of the LSB of the red channel in pixel
GreenMaskSize	Size of direct color green channel mask
GreenFieldPosition	Bit position of the LSB of the green channel in pixel
BlueMaskSize	Size of direct color blue channel mask
BlueFieldPosition	Bit position of the LSB of the blue channel in pixel
RsvdMaskSize	Size of direct color reserved channel mask
RsvdFieldPosition	Bit position of the LSB of the reserved channel in pixel

SV_palette

Structure describing a single palette entry used for programming the color palette in graphics modes. Note that this palette format is different to the standard format supported by the old VGA BIOS, but is the same as the VBE 2.0 and VBE/AF native formats.

```
typedef struct {
    uchar   blue;
    uchar   green;
    uchar   red;
    uchar   alpha;
    } SV_palette;
```

Declaration in

svga.h

Field	Description
blue	Blue component for color
green	Green component for color
red	Red component for color
alpha	Alpha or unused component for color (should always be zero!)

WD_event

Structure describing the information contained in an event extracted from the event queue.

```
typedef struct {
    uint        what;
    ulong       when;
    int         where_x;
    int         where_y;
    ulong       message;
    ulong       modifiers;
    } WD_event;
```

Declaration in

wdirect.h

Field	Description	
what	Type of event that occurred. Will be one of the following values:	
	Value	**Meaning**
	EVT_NULLEVT	No event has occurred
	EVT_KEYDOWN	A key has been pressed
	EVT_KEYREPEAT	A key is repeating while held down. Note that these events also have an associated repeat count so that the buffer does not fill up with mutiple repeat codes.
	EVT_KEYUP	A key has been released
	EVT_MOUSEDOWN	A mouse button has been pressed
	EVT_MOUSEUP	A mouse button has been released
	EVT_MOUSEMOVE	The mouse cursor has been moved
	EVT_TIMERTICK	A timer tick event has occured. The WD_setTimerTick function is used to set the interval between timer tick event
	EVT_USEREVT	All application defined events start with this number and go up from there

when	Time that the event occured in milliseconds since startup
where_x	X coordinate of the mouse cursor location at the time of the event (in screen coordinates)
where_y	Y coordinate of the mouse cursor location at the time of the event (in screen coordinates)
message	Event specific message for the event. For use events this can be any user specific information. For keyboard events this contains the ASCII code in bits 0-7, the keyboard scan code in bits 8-15 and the character repeat couont in bits 16-30. You can use the WD_asciiCode, WD_scanCode and WD_repeatCount macros to extract this information from the message field. For mouse events this contains information about which button was pressed, and will be a combination of the following flags:

Value	Meaning
EVT_LEFTBMASK	The left mouse button was down
EVT_RIGHTBMASK	The right mouse button was down
EVT_BOTHBMASK	Both mouse buttons were down (this is a combination of EVT_LEFTBMASK and EVT_RIGHTBMASK)
EVT_ALLBMASK	All buttons were down (for the moment this is the same as EVT_BOTHBMASK)

modifiers	Contains additional information about the state of the keyboard shift modifiers (Ctrl, Alt and Shift keys) when the event occured. For mouse events it will also contain the state of the mouse buttons. Will be a combination of the following flags:

Value	Meaning
EVT_LEFTBUT	The left button was down (for mouse events only)
EVT_RIGHTBUT	The right button was down (for mouse events only)
EVT_CTRLSTATE	One of the control keys was down
EVT_ALTSTATE	One of the alt key was down
EVT_SHIFTKEY	One of the shift key was down
EVT_DBLCLK	The mouse down event was a double click

Redistributable Components

Subject to the terms and conditions of the SciTech Software License Agreement, in addition to any Redistribution Rights granted therein, you are hereby granted a non-exclusive, royalty-free right to reproduce and distribute the Components specified below provided that (a) is distributed as part of and only with your software product; (b) you not suppress, later or remove proprietary copyright notices contained therein; and (c) you indemnify, hold harmless and defend SciTech Software and it's suppliers from and against any claims or lawsuits, including attorney's fees, that arise or result from the use or distribution of your software product.

The redistributable SuperVGA Kit and WinDirect components are:

- SVGA60.DLL (c) 1996 SciTech Software.
- SVGA60F.DLL (c) 1996 SciTech Software.
- PMPRO60.DLL (c) 1995-6 SciTech Software.
- PMPRO60F.DLL (c) 1995-6 SciTech Software.
- WDIR60.DLL (c) 1995-6 SciTech Software.
- WDIR60F.DLL (c) 1995-6 SciTech Software.
- DVA.386 (c) 1995 Microsoft Corporation

Software License Agreement

BEFORE PROCEEDING WITH THE INSTALLATION AND/OR USE OF THIS SOFTWARE, CAREFULLY READ THE FOLLOWING THE TERMS AND CONDITIONS OF THIS LICENSE AGREEMENT AND LIMITED WARRANTY (The "Agreement")..

BY INSTALLING OR USING THIS SOFTWARE YOU INDICATE YOUR ACCEPTANCE OF THIS AGREEMENT. IF YOU DO NOT ACCEPT OR AGREE WITH THESE TERMS, YOU SHOULD NOT INSTALL OR USE THIS SOFTWARE.

LICENSE

This software, including documentation, software and/or additional materials (the "Software") is owned by SciTech Software or by its suppliers and is protected by copyright law and international treaty provisions. This Agreement does not provide you with title or ownership of Product, but only a right of limited use. Upon payment of the proscribed license fee, the Software is hereby licensed to you (and not sold) by SciTech Software or a licensee of SciTech Software (in either case the entity licensing the Software to you is referred to in this Agreement as SciTech Software). SciTech Software hereby grants you a non-exclusive license to use the Software as set forth below:

Upon purchase of a license for the Software, you may:

- use the software only on one computer
- make one copy of the Software excluding the documentation for archival or backup purposes.
- integrate the Software with your Applications in Executable Code form only, subject to the redistribution terms below.
- permanently transfer your rights hereunder, provided you also transfer all copies of the Software and the documentation and provided the transferee agrees to the terms and conditions of this Agreement.
- If this is an "Evaluation Version" of the Software you may freely distribute complete package as released by SciTech Software to others.

REDISTRIBUTION RIGHTS

Upon purchase of a license for the Software, you are also granted a non-exclusive, royalty-free right to reproduce and redistribute executable files created using the Software (the "Executable Code") in conjunction with software products that you develop and/or market (the "Applications"). You may be granted additional rights to reproduce and distribute components of the Software. Such additional rights, including any restrictions thereto, and the components, shall be specified in the "Getting Started" document included in the Software documentation and in the invoice for the Software that you receive from SciTech Software. If a limitation is placed on the maximum number of copies that can be distributed under this Agreement, shareware and demo products created using the Software shall not be counted in the computation of that maximum number of copies.

RESTRICTIONS

With out the expressed, written consent of SciTech Software, you may **NOT**:

- use, copy, modify, or transfer the Software or the documentation, or any copy in whole or in part, except as expressly provide for in this license.
- allow the Software to be used on more than one computer at a time or by more than one person at a time except as otherwise expressly provided herein.
- rent or lease the Software.
- modify or adapt the Software in whole or in part including, but not limited to, translating or creating derivative works.
- disassemble or reverse compile for the purpose of reverse engineering the Software.
- defeat or try to defeat any user messages or copyright notices that the Software displays including, but not limited to, Evaluation Version notices and copyright notices.
- Sell any portion of the Product on its own, without integrating it into your Applications as Executable Code.
- Use the Software in the development an operating system, online service or their associated drivers & utilities, or in a graphic library.
- You may not distribute WinDirect components in conjunction with any of SciTech Software's DOS OEM bundle versions of the UniVBE/UniLib products. There are Windows-specific, equivalent versions of those products available from SciTech Software for distribution which are available for that purpose.

EVALUATION VERSIONS AND EVALUATION PERIOD

In order to allow you to evaluate the Software before you purchase it, SciTech has developed "Evaluation Versions" of the Software. Evaluation Versions will generally contain a message indicating that you are not allowed to distribute the Software with your Applications until you license a full copy of the Software from SciTech Software. You must comply with any restrictions contained in those messages or other documentation contained in the Evaluation Version. You may use the Evaluation Version for a period of up to 21 days, in order to determine whether the Software meets your needs before purchasing it. Your 21-day evaluation period begins when you first install the Software on one or more computers for evaluation purposes. Once the evaluation period ends, you agree to either purchase a legal copy of the Software, or to stop using it. If you have ordered a legal copy of the Software from us, you may continue to use the Evaluation Version until your legal copy arrives. While you are using an Evaluation Version, you may use it on as many computers as are required to perform your evaluation. Once the evaluation period is over and you purchase the Software, you may only use the Software on one computer at a time. F You must not defeat, or try to defeat, messages in the Software, including those which mark the software as an Evaluation Version.

You may make copies of a full, Evaluation Versions to give to others, however you may not sell Evaluation Versions for a profit (shareware distribution companies may charge their normal shipping and handling fees). Please also note that distribution of the Evaluation Versions may only be through the normal shareware distribution channels as a single, complete package.

CONFIDENTIALITY OF SOURCE CODE

You agree to maintain in confidence any source code portions of the Software by using at least the same physical and other security measures as you use for your own confidential technical information and documentation. You further agree not to disclose any source code portions of the Software, or any aspect thereof, to anyone other than employees or contractors who have a need to know or obtain access to such information in order to support your authorized use of the Software and are bound to protect such information against any other use or disclosure. These obligations shall not apply to any information generally available to the public, independently developed or obtained without reliance on SciTech Software's information, or approved for release by Licensor without restriction.

TERMINATION

Should you fail to carry out any other obligation under this Agreement or any other agreement with SciTech Software, SciTech Software may, at its option, in addition to other available remedies, terminate this Agreement immediately upon written notice. Your license will terminate, and you are required to return or destroy, as requested by SciTech Software, all copies of the Product(s) in your possession (whether modified or unmodified), and all other materials pertaining to the Product(s), including all copies thereof. You agree to certify your compliance with such requirement upon SciTech Software's request. All disclaimers of warranties and limitation of liability set forth in this Agreement shall survive any termination of this Agreement.

SELECTION AND USE

You assume full responsibility for the selection of the Software to achieve your intended results and for the installation, use and results obtained from the Software.

LIMITED WARRANTY

THIS SOFTWARE IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PRODUCT IS WITH YOU. SHOULD THE PRODUCT PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING OR ERROR CORRECTION.

SCITECH SOFTWARE DOES NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE SOFTWARE WILL MEET YOUR REQUIREMENTS OR THAT THE OPERATION OF THE SOFTWARE WILL BE UNINTERRUPTED OR ERROR FREE.

No oral or written information given by SciTech Software, its agents or employees shall create a warranty.

LIMITATION OF REMEDIES AND LIABILITY.

IN NO EVENT SHALL SCITECH SOFTWARE, OR ANY OTHER PARTY WHO MAY HAVE DISTRIBUTED THE SOFTWARE AS PERMITTED ABOVE, BE LIABLE FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE SOFTWARE (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR FAILURE OF THE SOFTWARE TO OPERATE WITH ANY OTHER PRODUCTS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The cumulative liability of SciTech Software to you for all claims relating to the Software, in contract, tort, or otherwise, shall not exceed the total amount of license fees paid to SciTech Software for the relevant Software. The foregoing limitation of liability and exclusion of certain damages shall apply regardless of the success or effectiveness of other remedies.

GENERAL

Notices. All notices or other communications required to be given shall be in writing and delivered either personally or by U.S. mail, certified, return receipt requested, postage prepaid, and addressed as provided in the Agreement or as otherwise requested by the receiving party. Notices delivered personally shall be effective upon delivery and notices delivered by mail shall be effective upon their receipt by the party to whom they are addressed.

Severability. Should any term of this License Agreement be declared void or unenforceable by any court of competent jurisdiction, such declaration shall have no effect on the remaining terms hereof.

Governing Law. This Agreement shall be governed by and construed and enforced in accordance with the laws of the State of California, USA as it applies to a contract made and performed in such state.

No Waiver. The failure of either party to enforce any rights granted hereunder or to take action against the other party in the event of any breach hereunder shall not be deemed a waiver by that party as to subsequent enforcement of rights or subsequent actions in the event of future breaches.

Costs of Litigation. If any action is brought by either party to this License Agreement against the other party regarding the subject matter hereof, the prevailing party shall be entitled to recover, in addition to any other relief granted, reasonable attorney fees and expenses of litigation.

If you have any questions regarding this agreement, please contact SciTech Software at (916)-894-8400.

Index

Z