

SolVBE 1.3b

Universal VESA VBE driver for Windows2000/Windows XP

<http://solvbe.sourceforge.net/>

Jari Komppa

<http://iki.fi/sol/>

Table of Contents

1	Explanation of Abbreviations.....	3
2	Disclaimer.....	3
3	Program Usage.....	4
3.1	What is SolVBE?.....	4
3.2	How to Use SolVBE.....	5
3.3	Command-Line Options.....	5
3.4	What Should Work, and What Doesn't.....	10
3.4.1	Current Feature Set.....	10
3.4.2	Known Issues.....	10
4	Background.....	11
4.1	Motivation.....	11
4.2	The Future.....	11
5	Technical Overview.....	12
5.1	Subsystems.....	12
5.2	Interrupt Services.....	13
5.3	Video Setup.....	14
5.4	Display Refresh.....	15
5.5	Mouse Callback.....	16
6	The SolVBE TSR.....	17
6.1	Initialization.....	17
6.2	Inline Assembly.....	17
6.3	Interrupt handlers.....	18
6.4	Assembler.....	18
7	The SolVBE VDD.....	19
7.1	VDD Source File Structure.....	19
7.2	Initialization.....	19
7.3	TSR - VDD Communication.....	19
7.4	Video Calls.....	20
7.4.1	The VGA Ports.....	20
7.4.2	The INT10h Handler.....	20
7.5	VESA VBE calls.....	21
7.5.1	VESA Information.....	21
7.5.2	VESA Mode Information.....	21
7.5.3	VESA Mode Setup.....	21
7.5.4	VESA Mode Query.....	21
7.5.5	VESA Bank Switching.....	21
7.5.6	VESA Scan Line Length Control.....	22
7.5.7	VESA Display Offset.....	22
7.5.8	VESA DAC Control.....	22
7.5.9	VESA Palette Control.....	22
7.6	Mouse Calls.....	22
7.6.1	Capturing and Releasing the Mouse.....	23
7.6.2	Talking Back to the Mouse Driver.....	23
7.6.3	The INT33h Handler.....	23
7.7	User Interface and the Frame Buffer.....	23
7.7.1	Worker Thread.....	23
7.7.2	Window Initialization.....	23
7.7.3	Frame Buffer Window Messages.....	24
7.7.4	Video Memory.....	24
7.7.5	Frame Buffer.....	24
7.7.6	Rendering.....	24

1 Explanation of Abbreviations

<i>Term</i>	<i>Description</i>
BIOS	Basic Input/Output System.
DDK	Driver Development Kit, required for VDD development.
DOS	Disk Operating System, precursor of Windows.
INT	Interrupt, in our context this means a system subroutine call in DOS.
IRQ	Interrupt Request, usually an interrupt caused by a hardware event.
TSR	Terminate and Stay Resident, a form of DOS driver.
VBE	VESA BIOS Extension, standard video BIOS extension.
VDD	Virtual Device Driver, an extension mechanism for VDM.
VDM	Virtual Dos Machine, a subsystem in Windows2000/WindowsXP that acts as a compatibility layer towards old DOS programs.
VESA	Video Electronics Standards Association, organization that promotes standards in computer video electronics.
VGA	Video Graphics Array, a de-facto standard video card on PCs popularized by IBM.
VOGONS	Very Old Games On New Systems, an internet community that attempts to find and promote ways to run old programs on modern systems.

2 Disclaimer

No guarantees. Use at your own risk. If it breaks, you have the pieces.

SolVBE is considered to be in beta. It saves a log file (called SolVBE.log, surprisingly) in the root of the C drive.

If you have problems, feel free to contact me; my current email address can be found at <http://iki.fi/sol>. If contacting for support reasons, please include:

- description of your system (OS version, video card, cpu speed),
- description of the problem (what you're trying to do, what happens)
- if it feels relevant, SolVBE.log

Another good place to ask questions is VOGONS at <http://vogons.zetafleet.com>. Please DO NOT post solvbe.log there, unless asked.

3 Program Usage

3.1 What is SolVBE?

SolVBE is a VESA VBE 1.2 driver for Windows 2000 / Windows XP command prompt.

What this means is that after running SolVBE, you will be able to run some old DOS programs that require VESA support, even if your display adapter doesn't support VESA bios extensions under Windows.

Additionally it enables you to run such programs in windowed mode. In addition to VESA modes, standard VGA 13h mode (320x200x256c) is also supported, and there is some limited support for 'tweaked modes' as well.

Running old programs under windows is a complicated task, and SolVBE only solves one piece of the puzzle. Other reasons for old programs not to work include:

- Audio problems
 - GusEmu (<http://listen.to/gusemu>) provides Gravis Ultrasound emulation
 - VDMSound (<http://vdmsound.sourceforge.net>) provides SoundBlaster emulation (among other things)
- cli/popf problem (application freezing due to an unfortunate design flaw in the x86 virtual real mode)
 - CLI2NOP (<http://vogons.zetafleet.com/viewtopic.php?t=31>) attempts to patch executables to work around this problem.
- Conflicts with some other windows subsystems.
 - Dosbox (<http://dosbox.sourceforge.net>) is a x86 virtual machine, capable of running old DOS programs, but requires more CPU power. Contains some workarounds for badly behaving DOS programs.
 - Qemu (<http://fabrice.bellard.free.fr/qemu/>) is another x86 virtual machine, faster and more "pure" virtual machine than dosbox is, but is more difficult to use.
- The original code may simply be so badly behaving that it shouldn't have worked in the first place.

In general, dosbox is the best overall solution – it usually works even if everything else fails. It's only bad side is its relatively slow speed, but that is being worked on.

3.2 How to Use SolVBE

1. Copy SolVBE into your application's directory.
2. Launch SolVBE
3. Launch the application

You may wish to create a batch file that launches SolVBE before launching your application. Some applications already use batch files to launch; in this case it is enough to add SolVBE at the beginning.

The batch file will be especially helpful if you need to add any command-line parameters to alter SolVBE's behavior.

Note that often both video and audio are broken, so it is recommended that you get VDMSound, and run the batch file by right-clicking on the batch file and select "run with vdmsound". (Note that VDMSound's 'vesa mode support' may conflict with SolVBE. If you're having problems, make sure it's turned off).

When the application is running, you can use the middle mouse button to give the application the mouse focus. Click middle mouse button again to remove the focus. Right mouse button can be used to open the pop up menu, which can be used to configure many of SolVBE's options at runtime.

3.3 Command-Line Options

In case you're experiencing problems (or simply want to change SolVBE's default behavior), you can run SolVBE with one or more of the following command-line options:

Parameter Description

- m0** *Disable mouse callback support.*
This option will disable the mouse callback support. Some applications may work without it.
- m1** *IRQ12 capture mode.*
This mode hooks the IRQ 12, which is typically reserved for the PS2 mouse. This IRQ is triggered whenever mouse moves or a mouse button is pressed.
This mode is the default.
- m2** *IRQ12 chain mode.*
This mode works somewhat like the above, except that it chains the interrupt instead of capturing it. Thus, other things hooked to IRQ 12 might work.
- m3** *IRQ capture with callbacks at the rate of the vertical retrace.*
Try this mode if you think that the mouse event gets called too often.
- m4** *IRQ chain with callbacks at rate of the vertical retrace.*
Try this mode if you think that the mouse event gets called too often, and you'd be using -m2 otherwise.
- m5** *IRQ capture with a flood of IRQ calls.*
Try this if you think that the mouse event doesn't get called often enough. (somewhat experimental)
- m6** *IRQ chain with a flood of IRQ calls*
Try this if you think that the mouse event doesn't get called often enough, but you'd be using -m2 otherwise. (somewhat experimental)
- m8** *IRQ0 chain mode*
In this mode, the mouse events are sent whenever a timer interrupt occurs. This mode relies on applications reprogramming the timer to run at a very fast rate.
Try this first if -m1 doesn't work for you.
- p0** *Minimal VGA register simulation*
This option disables the tweaked mode support. May help of harm in miscellaneous VGA problems.
- p1** *Normal video operation*
In this mode, vertical retraces occur approximately 60 frames per second, and tweak mode support is enabled.
This mode is the default.
- t0** *Disable high performance counter*
In case the high performance timer code causes the frame rate to be extremely slow (and vertical retraces seem to take forever), this disables the use of the high performance counter. Frame lengths will not be (even) as accurate as in the normal mode; log timings will be off.
- t1** *Enable high performance counter*
This mode is the default
- f0** *Disable linear filtering*
This option can be changed at runtime.

Parameter Description

- f1** *Enable linear filtering*
This option can be changed at runtime.
This mode is the default.
- a0** *Disable aspect ratio locking*
This option can be changed at runtime.
- a1** *Enable aspect ratio locking*
This option can be changed at runtime.
This mode is the default.
- x0** *Disable 16-bit textures*
This mode is the default.
- x1** *Enable 16-bit textures*
This option may help with performance, as it asks for OpenGL to use 16-bit textures instead of whatever is the default. This will degrade the visual quality a bit, though, and whether it helps with performance depends on your system.
- w0** *Don't assume video memory to be write-only*
This mode is the default.
- w1** *Assume video memory to be write-only*
This option lets SolVBE assume that all video memory gets overwritten all the time, and thus does not bother copying data back "down" whenever VESA banks or tweak mode planes change. If you get garbled displays with this option on, this assumption is wrong for that particular application.
- r0** *Don't assume read mode planes are read-only*
This mode is the default.
- r1** *Assume read mode planes are read-only*
This option lets SolVBE assume that whenever the application asks for a VESA bank or tweaked mode plane and says that it's for reading purposes, it won't get modified. If you get garbled displays with this option on, this assumption is wrong for that particular application.
- d0** *Refresh as fast as possible*
- d1** *Refresh at 5hz*
- d2** *Refresh at 10hz*
- d3** *Refresh at 15hz*
- d4** *Refresh at 20hz*
- d5** *Refresh at 25hz*
- d6** *Refresh at 30hz*
- d7** *Refresh at 40hz*
- d8** *Refresh at 50hz*
- d9** *Refresh at 60hz*
- da** *Refresh at 80hz*

Parameter Description

-db *Refresh at 100hz*

These options set the desired refresh rate of the display. The higher the refresh, the more SolVBE will be wasting time updating the screen. If your game isn't running smoothly, set the refresh rate lower.

-v0 *Vertical Retrace is flip-flop*

-v1 *Vertical Retrace at 5hz*

-v2 *Vertical Retrace at 10hz*

-v3 *Vertical Retrace at 15hz*

-v4 *Vertical Retrace at 20hz*

-v5 *Vertical Retrace at 25hz*

-v6 *Vertical Retrace at 30hz*

-v7 *Vertical Retrace at 40hz*

-v8 *Vertical Retrace at 50hz*

-v9 *Vertical Retrace at 60hz*

-va *Vertical Retrace at 70hz*

-vb *Vertical Retrace at 80hz*

-vc *Vertical Retrace at 100hz*

-vd *Vertical Retrace at 120hz*

These options control the rate of the vertical retrace simulation. If your application is vertical retrace limited, you may be able to alter the application's speed with this option. Set the rate to -v0 for fastest possible operation - in that mode, every second time the application asks for vertical retrace state, it will be on or off.

-s0 *Don't sleep at Vertical Retrace*

-s1 *Sleep(0) at Vertical Retrace*

-s2 *Sleep(1) at Vertical Retrace*

-s3 *Sleep(2) at Vertical Retrace*

-s4 *Sleep(3) at Vertical Retrace*

-s5 *Sleep(4) at Vertical Retrace*

-s6 *Sleep(5) at Vertical Retrace*

-s7 *Sleep(6) at Vertical Retrace*

-s8 *Sleep(7) at Vertical Retrace*

-s9 *Sleep(8) at Vertical Retrace*

-sa *Sleep(9) at Vertical Retrace*

-sb *Sleep(10) at Vertical Retrace*

These options control the amount of sleep performed during vertical retrace period. You can use this to make sure that the application gets done whatever it wants to do during the vertical retrace period. Setting this to -s0 will most likely hang at "waiting for retrace".

-l0 *Set debug level to 'none'*

Ignore noncritical errors, no logging.

-l1 *Set debug level to 'warn'*

Display warnings, no logging.

Parameter Description

- l2 *Set debug level to 'log'*
Display warnings and write log.
This mode is the default.
- l3 *Set debug level to 'extended logging'*
Display extended warnings and write log.

3.4 What Should Work, and What Doesn't

3.4.1 Current Feature Set

- Large part of VESA VBE 1.2 specification implemented
- VGA 13h support
- 320x200 and 320x400 un-chain4 tweak mode support (no page flipping support though)
- VGA palette port registers support
- Mouse support
- OpenGL scaling output
- Bilinear filtered and nearest-pixel modes
- Full-screen mode (uses 640x480x32 always)
- Optional 4:3 aspect ratio locking

3.4.2 Known Issues

- 16-color modes won't work.
- Vertical retrace simulation is somewhat broken (again). Timing palette changes to the vertical retrace doesn't work (e.g. In 'death rally' intro movie).
- SolVBE seems to require approximately 2GHz CPU to run. Slower systems seem to hang, so it is possible that there is a racing condition somewhere.

4 Background

4.1 Motivation

SolVBE started from my frustration of getting Terra Nova: Strike Force Centauri running under Windows XP. Some people managed to get TN:SFC running under XP with some hacky VBE "drivers" which simply disabled VESA VBE 2.0 linear frame buffer support, and thus forced Terra Nova to use the VESA 1.2 interface instead.

Unfortunately (or fortunately, depending on your point of view), I couldn't get VESA 1.2 modes working with my display adapter (Radeon 9700 Pro) under Windows XP. Either the VESA BIOS on the card is broken, or windows conflicts with it, or the display adaptor's own drivers block it in some way. In any case, I was stuck.

My options were either to run the game under dosbox (a DOS virtual machine), or to wait for VDMSound (soundblaster emulator for NT console) to get VESA support.

Dosbox is the way to go, eventually, but currently Terra Nova is way too heavy for it. Even on a 2.6GHz P4, the game literally crawls, before crashing. VDMSound's latest beta versions have an option saying something about VESA support, but there was no way of telling when or if the support appears.

The alternative was to take the plunge and write my own VESA driver.

So I got my hands on the windows DDK, downloaded OpenWatcom 1.2, and started hacking. Spending my weekends and evenings on the project, in a week and a half (February 24 - March 4, 2004) I finally could play TN. Another four days later I had my first full featured beta running and posted on VOGONS.

After that I added simple tweaked mode support (which, unfortunately, doesn't help with Terra Nova due to the way it uses the mode), and tons of tweaking options so that people who have problems can try to get things working.

SolVBE has since been used by different people to play old games on new systems, and to play some old games in a window that would normally require full-screen mode. It has also been used to run some old scientific software under windows.

4.2 The Future

The most requested (and most useful) extension to SolVBE would be full VESA 2.0 linear frame buffer support. This would require some more research, as it would need at least protected mode interrupt handlers (the same way as the TSR does real-mode int10h handling), as well as research on how the physical/virtual memory mapping would work in windows world.

There is no reason why SolVBE wouldn't work on slower machines in theory, but in practise it requires rather powerful CPU. I do not currently know why.

Further tweaked mode support would enable people to play many tweaked mode games in windowed modes, as well as enabling some old demos.

Other VESA VBE standards could also be implemented (such as VBE/AF), but I'm not aware of any software that actually uses those.

5 Technical Overview

5.1 Subsystems

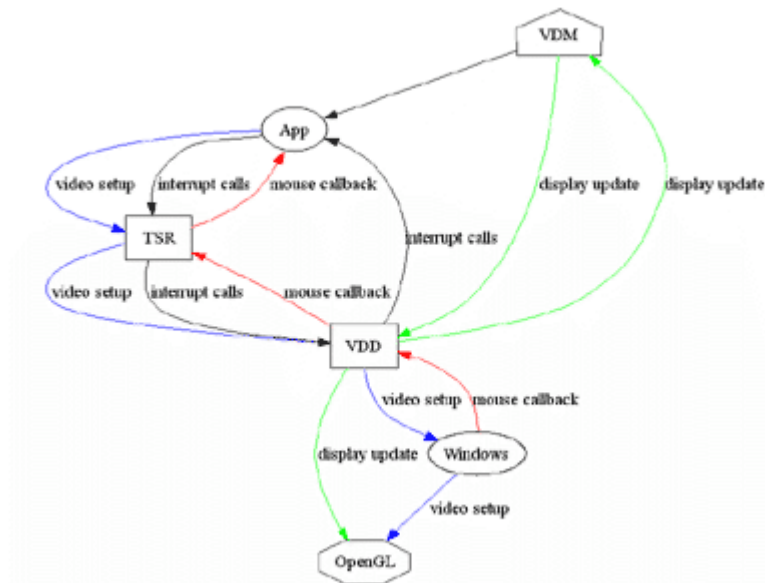


Illustration 1 Inter-subsystem calls.

SolVBE comes in two parts: a 16-bit DOS TSR, which traps the DOS interrupts, and a 32-bit windows VDD, which does the actual work.

The TSR captures interrupts 10h (video services), 33h (mouse services) and optionally either IRQ12 (PS2 mouse) or IRQ0 (timer). The mouse capture is needed since the VDM thinks that we're still in text mode, and thus cannot offer its mouse support.

Additionally, since we're stealing focus from the console, the keyboard events need to be handled. This is done by simply forwarding keyboard events to the console.

The VDD traps several VGA ports. The DDK explicitly says that one should not do that, but it just happens to work right now. It's rather likely that future Windows releases (or even service packs) might render SolVBE unusable because of this.

The ports are used for three things: palette control, vertical retrace simulation, and tweak modes.

Without the palette control, SolVBE wouldn't be of much use. Although VESA interface includes palette setting functions, pretty much all applications program palette directly through VGA ports.

Vertical retrace simulation is important for two reasons; first being speed (for applications that are frame-bound instead of real time clock-bound), and second being palette synchronization. If vertical retrace isn't simulated properly, some applications will show wrong colors causing irritating blinking to occur.

Tweak modes are achieved by reprogramming the VGA timings. Several different resolutions and refresh rates can be achieved, most famous being 'mode x', 320x240x256c mode, which also supports page flipping. The other often used mode is 320x400x256c. Since these modes use more memory than is available in the video segment (64k), these modes are usually 'planar', meaning that you can only access 1/4th of the memory at a time, but you can switch between the 'planes' you wish to access.

At the time of this writing (1.3b), the tweaked mode support is pretty basic, and only supports a single-page 320x400 mode.

5.2 Interrupt Services

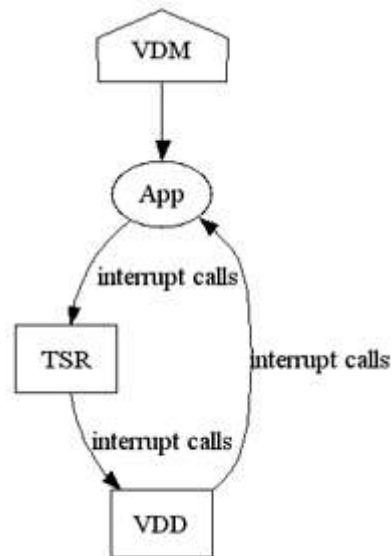


Illustration 2 Interrupt Services

Whenever the DOS application performs an interrupt call, say, to query the current mouse position from the mouse driver via int33h, the TSR catches this call, forwards it to the VDD, which modifies registers in the TSR's stack, and optionally writes something directly into the application's memory, and then returns access back to the application.

The application never knows that we've just popped into windows protected mode world for a while, and then back. From DOS-era point of view, this is all extremely cpu-power wasting, but we can afford it now.

The TSR traps INT10h for video services and INT33h for mouse services. The actual implementation of the interrupt handlers exists in the VDD. Additionally, IRQ12 or IRQ0 is captured so that the VDD can interrupt the VDM whenever a mouse event occurs.

5.3 Video Setup

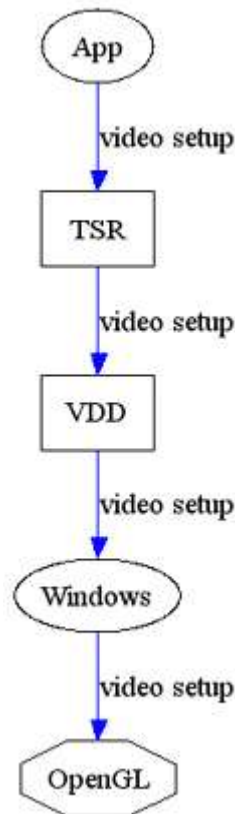


Illustration 3 Video setup call chain

SolVBE uses the following buffers for the image data:

- Video segment (real-mode a000)
- Video memory buffer (Windows GDI bitmap)
- Frame buffer (Windows GDI bitmap)
- OpenGL texture

The video segment and OpenGL texture are no-brainers, but why use Windows GDI bitmaps for video memory, and why is frame buffer different?

First, GDI bitmaps support 4, 8, 15, 16, 24 and 32 bit modes. By using those, we don't need to perform any pixel conversions ourselves; everything "just works". Replacing these with specialized code that just performs the conversions would most likely bring some performance gain, but I don't know if it's worth it.

The frame buffer is pretty much the same thing - pixel conversion. OpenGL's texture format has a different RGB ordering than windows bitmaps generally do, so I'm killing two birds with one stone by blitting the video memory to the frame buffer - first, the pixel conversion is done, and second, byte order is changed to be OpenGL friendly.

(As it happens, 4bit modes do not work, and would require specialized handling. As I haven't heard of any need for 16-color modes, the 4-bit support is not a priority).

5.4 Display Refresh

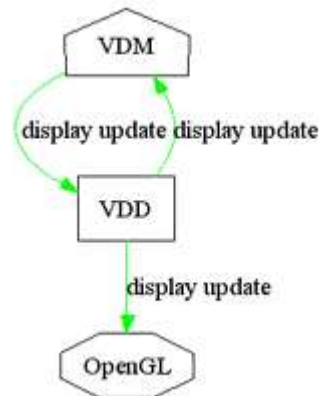


Illustration 4 Display Refresh Loop

The VDD runs a worker thread which also takes care of the frame buffer window. This thread runs in a loop, and periodically copies memory from the real-mode video segment (0xa000) to its video memory buffer, and then transfers the data to an OpenGL texture, and finally renders this on screen using two triangles.

Additionally, whenever the DOS application changes VESA banks (or tweak mode planes), we need to copy the memory from the video segment into our video memory buffer, and we need to copy data back to the video segment from the new location in the video memory buffer.

This is because we have no control over what the application does with the video memory. In the worst-case scenario (which happens with Terra Nova: Strike Force Centauri, actually), the application switches tweak mode planes after every single pixel, and thus for a 320x400 screen, we need to transfer a whopping 16777216000 bytes (about 16GB) per frame (or, if we desire 20Hz framerate, 312,5GB/s).

5.5 Mouse Callback

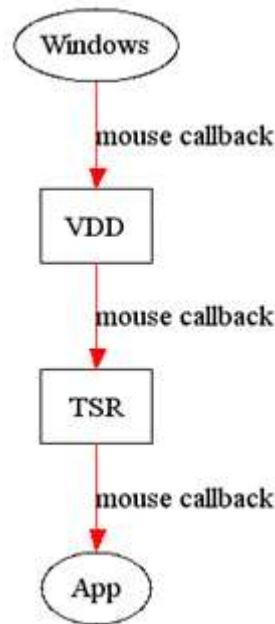


Illustration 5 Mouse callback

One unfortunate "feature" in the DOS mouse driver is the mouse callback. It's one of these things that were good ideas at the time, but simulating it on modern machines is a major headache.

Basically, the application registers a function which needs to be called whenever the mouse moves or a button is pressed or released. In order to make this happen, the application flow must be interrupted and this call made. On a real dos machine this is achieved via a hardware interrupt. In our case, we simulate IRQ12 hardware interrupt, and have a tiny piece of code in the TSR which passes the call along to the application.

Unfortunately things don't seem to be this simple. On some applications the mouse movement is very jerky, while others work flawlessly. Several options were added in 1.2b, including the use of IRQ0 (system timer) instead of IRQ12, and interrupt flooding (where we cause IRQ12 to occur *a lot*). These are not real fixes, but may help in some cases.

6 The SolVBE TSR

The TSR portion of SolVBE is relatively simple. Its task is to capture video and mouse interrupt events from applications and pass the events on to the VDD.

OpenWatcom 1.2 is required to compile the TSR.

Due to its nature, the TSR includes a fair portion of assembler. Still, all of the assembler functions are pretty simple.

The whole TSR could be written as a .COM file instead of an .EXE, which would make some things easier, as well as freeing more conventional memory.

6.1 Initialization

The TSR entry point is the standard main() function.

At initialization, the TSR attempts to register the VDD .dll file, and reports error if failed.

After that, it reports bank switch procedure, info string, mode table and 'mouse trap' offsets to the VDD.

Next, the command line parameter list is passed on to the VDD. The VDD returns the desired operation mode (IRQ 12 capturing or chaining, or IRQ 0 chaining, or no IRQ capture at all).

Interrupts 10h and 33h are captured, and finally _dos_keep() function is called to make the program resident.

6.2 Inline Assembly

The TSR implements several things as inline assembly.

VDDRegisterModule() attempts to initialize the VDD .dll.

VDDUnregisterModule() deinitializes the VDD .dll. Since there is no way to unload the TSR, this function is not actually used.

VDDDispatchCall() is the primary communication channel between the TSR and the VDD.

VDDUnsimulate16() is the counterpart of the VDD's VDDSimulate16() call. However, SolVBE doesn't currently use the VDDSimulate16().

int_setXX() can be used to manipulate registers in the application's stack while inside an interrupt handler. All such manipulation is currently done inside the VDD, so these functions exist mainly for debugging purposes.

mousecallback() performs some register juggling to report mouse position to the application without destroying the application's critical register values.

Eoi() performs the End of Interrupt, which signals the end of an IRQ call.

6.3 Interrupt handlers

The `videoint()` and `mouseint()` functions are the handlers for INT10h and INT33h. Both simply pass control to the VDD, and call the old interrupt handlers if the VDD returns a non-zero value.

The `mouseIRQ()` handler does the same, except that if a non-zero value is returned, it calls the `mousecallback()` inline assembler function to report the mouse position to the application. Finally, it calls `eoi()` to end the IRQ handling.

The `mousecbint()` handler performs the same functionality as above, except that instead of `eoi()`, it calls the old IRQ handler.

6.4 Assembler

Some bits could not easily be made in inline assembly, so they were placed in an assembler source file.

`gMouseTrap` is the far call jump target for the mouse handler.

`bankswitchproc()` is one function the applications can call directly to bypass interrupt overhead when switching video banks. Since we're not interested in such small performance gains, we're simply calling the video interrupt here.

Main reason for placing these things here was that the Watcom compiler kept messing up some of the registers. Some things done in inline assembly might be simpler if just implemented here.

The "mouse trap" has to be in the code segment, and the easiest way to put it there was though the use of the assembler file.

7 The SolVBE VDD

The VDD portion of SolVBE performs most of the functionality. In fact, no changes have been needed in the TSR portion since 1.1b.

Visual Studio 6.0 or later with Windows 2000 DDK is needed to compile the VDD.

7.1 VDD Source File Structure

The VDD source code has been split into the following units:

<i>Source File</i>	<i>Contents</i>
SolVBE.h	Includes, definitions, structures, enumerations, externs.
SolVBE.cpp	Global variables, VDD interface, command line parsing, logging, miscellaneous.
vesa.cpp	VESA VBE calls (called from int10h).
display.cpp	Frame buffer, video memory buffer and window related functions, including pop-up menu handling and the worker thread.
int10h.cpp	INT10h handler, VGA registers
int33h.cpp	INT33h handler, mouse routines

7.2 Initialization

The DLL entry point is DllMain(), which is in the SolVBE.cpp file. The function is called when the TSR initializes the .dll file.

First, the high-performance counter is initialized, followed by reset of the log. (This much of the logging is performed regardless of the command line options, as they are not handled at this point).

This is followed by other critical initializations: mapping of the video segment, setup of the critical section that will be used for thread synchronization, and thread creation. The palette is set to random initial values.

After the DLL is initialized, the TSR calls the VDD to configure some values (see below on TSR – VDD communication for details on the values).

As a part of the initialization, the VDD has to find the console window's window handle. The official way of doing this is to set the console's title to a known value, and ask for windows to find the window with said value. This behavior is implemented in the findConsoleWindow() function.

Command line is parsed in the parseCmd() function.

7.3 TSR - VDD Communication

The TSR – VDD communication can seem somewhat confusing, as there are actually three different sets of CPU registers in use. The topmost, i.e. the registers that are in active use when VDD is running, are not accessed directly from C++ code.

The registers that are in active use by the TSR are altered by the VDD by using setAX(), setBX() etc. calls, and queried through getAX(), getBX() etc. calls. These functions are defined by the DDK.

The registers that are in use by the application that uses the TSR's services are altered through gRegs [AX], gRegs[BX], etc. variables. The gRegs array is mapped to the application's stack, to which all of the registers have been pushed before an interrupt call.

Most of the TSR – VDD communication runs through the SolVBECall() function. The TSR sets its BX, CX and DX register to some known values, and uses the VDM interface to invoke a call to SolVBECall().

The BX register is used to communicate what the call is about.

<i>BX value</i>	<i>Description</i>
0x0100	Configure call 1: data segment, info string offset, mode list offset.
0x0200	Configure call 2: Code segment, bank switch function offset, “mouse trap” offset.
0x0300	Configure call 3: Command line parameters.
0x0010	INT10h call.
0x0033	INT33h call.
0x0074	Mouse IRQ call.

Whenever INT10h or INT33h call is made, the gRegs array is mapped to the stack segment, thus allowing the VDD to alter the contents of the application's registers.

7.4 Video Calls

All of the video calls are in the int10h.cpp file.

An application has three ways of talking with the video hardware: First is using the INT10h interrupt, which is usually used for video card detection and video mode setup (as well as VESA VBE functionality). Second is the video memory, which is usually simply mapped to the 0xa000 segment. The third is the use of video registers. Even the simplest application uses video registers for two reasons: palette changes and vertical retrace checking.

7.4.1 The VGA Ports

The capturePorts() and releasePorts() functions are used to capture and release VGA ports. The capturePorts() is called whenever video mode is changed from text mode to something else, and releasePorts() is called when some text mode is set.

Whenever the ports are captured, the VDM calls vgareg_inb() and vgareg_outb() when the application accesses the VGA ports. The implementation of the vgareg_XXb() functions is more or less minimal at this point, only giving support for vertical retrace checks, palette manipulation and primitive tweaked mode support.

SolVBE does not attempt to simulate the VGA hardware; instead, it attempts to figure out what the application is trying to do, and does the most common thing in that case. Unfortunately many of the tweaked mode setups are extremely complicated (consisting of hundreds of register manipulations). Most of the complicated mode tweaks fail mostly because SolVBE does not return any sane values for the unsupported registers.

7.4.2 The INT10h Handler

The int10h handler (handle_int10()) does one of three things depending on the calling register values. Calls which are harmful and not supported (such as VESA 2.0 calls) are blocked. Supported calls are dispatched to the vesaXXXX() functions, and finally there are some interrupts which are not harmful but

which we have no reason to block, such as the console text output calls. These are passed through to the old int10h handler.

7.5 VESA VBE calls

All of the VESA VBE compatibility functions are defined in the vesa.cpp file.

7.5.1 VESA Information

When the application requests VESA information, the INT10h handler calls the vesainfo() function. This function maps the buffer that is inside the application's address space, and fills it out with legal values which correspond to the functionality SolVBE currently offers.

Additionally, the mode list that is within the TSR's address space is filled with legal values. The vesa info block contains a pointer to the mode list, as well as a pointer to VESA vendor information string (which also is contained within the TSR).

7.5.2 VESA Mode Information

Usually, applications that use VESA modes query through the VESA mode list, trying to find the one mode that they support. When the application asks for information about some mode, the INT10h handler calls vesamodeinfo().

The vesamodeinfo() function, much like the vesainfo() function, fills out the buffer in application's memory space with legal values about the queried mode. If an unknown mode is asked about, error is reported to the application.

7.5.3 VESA Mode Setup

When an application is satisfied with a mode, it sets it up, and INT10h handler calls vesasetmode(). This function is actually called regardless of whether the mode is a VESA mode or a VGA mode. The function calls deinitVideoMemory() and deinitFrameBuffer(), figures out the parameters for the new mode, and calls initVideoMemory() and initFrameBuffer().

Some VGA and mouse-related variables are reset, and, if the old mode was a text mode, VGA registers are captured.

7.5.4 VESA Mode Query

When the application asks for the current VESA mode, the INT10h handler calls vesagetmode(), which always returns success with the currently active mode number.

7.5.5 VESA Bank Switching

Since the application can only see 64k of the video memory at any one time, it will need to switch memory banks. When this happens, the INT10h handler calls vesamemorycontrol().

There we first copy the current contents of the a000 segment into the video memory buffer, and then we set the new segment offset. If we cannot assume the video memory to be write-only, we then need to copy the new bank down to the a000 segment.

7.5.6 VESA Scan Line Length Control

In order to save some processing power, some applications request a virtual screen with a scan line length of a power of two. Thus, for a 640x480x256c screen, the application may want the screen resolution to be 1024x480x256c instead. This way, they can update exactly 64 scan lines for each bank. When this happens, the INT10h handler calls `vesasetscanlinelength()`.

The function basically re-initializes the video memory to have the desired scan line length.

7.5.7 VESA Display Offset

If an application wants non-blinking display updates, it may use two “pages” of video memory and swap between them, only updating the page that is not currently visible. Less commonly the display offset might be used for smooth scrolling.

When the application wants to change the display offset, the INT10h handler calls `vesasetdisplaystart()`.

This is a rather cheap operation for us, as all we do is record the offset at which the video memory buffer should be blitted to the frame buffer.

7.5.8 VESA DAC Control

The standard VGA has a 6-bit DAC, which is only able to reproduce 18-bit color. Newer video hardware supported 8-bit DACs, but for compatibility reasons, the APIs only accepted 6-bit values. In order to be able to output full 8-bit values, the application would ask the VESA to accept those. When this happens, the INT10h handler calls `vesadaccontrol()`.

7.5.9 VESA Palette Control

If written exactly to the VESA specification, an application would only alter the palette through the VESA interrupt calls. However, since I haven't found a single application that would do so, the `vesasetpalette()` function is currently not implemented.

7.6 Mouse Calls

All mouse-related functions are in the `int33h.cpp` file.

If screen coordinates for the mouse were all that was needed, the mouse support would be rather simple. Unfortunately, most DOS applications use relative mouse coordinates instead, and thus we need to simulate the way DOS mouse drivers work.

When the mouse is captured, the real mouse cursor is set to the center of the frame buffer window. Each frame, if the mouse has moved, its movements are recorded, and then re-centered again.

7.6.1 Capturing and Releasing the Mouse

Since windows is a multitasking environment, we cannot steal the mouse outright. Instead, the mouse is captured and released whenever the user wants the mouse events to be sent to the DOS application. The `captureMouse()` function is called whenever the user clicks the middle mouse button (or mouse wheel) on the frame buffer window, and the mouse is freed with `releaseMouse()`, which gets called with another middle button click.

7.6.2 Talking Back to the Mouse Driver

If the application has used the “mouse driver callback” functionality, telling the mouse driver to call one of its local functions when the mouse moves (or a button is pressed), a complicated chain of event occurs.

First, we need to interrupt the VDM. This is done through IRQ12, and we simulate this interrupt with the `VDDSimulateInterrupt()` function. The function call is wrapped inside our `triggerMouseIRQ()` function.

Then, the TSR traps this IRQ, and calls the VDD, which figures out whether a new mouse motion call is required. If so, the TSR will call the application's callback function. The application's callback function pointer is kept in the `gMouseTrap` variable, which has to be kept inside the code segment, as we may not alter the data segment at all – otherwise, the application couldn't do anything sensible in the callback function.

7.6.3 The INT33h Handler

The application's INT33h calls are handled in the `handle_int33()` function.

The mouse interrupt handling is fairly complete. Unsupported calls are blocked.

7.7 User Interface and the Frame Buffer

All user interface and frame buffer related functionality can be found in the `display.cpp` file.

7.7.1 Worker Thread

The worker thread runs inside the `threadproc()` function.

When the thread starts, it calls `initWindow()` to initialize the frame buffer window.

While the thread is alive, it periodically updates the video memory buffer from the `a000` segment, calls `render()` to update the window, simulates the vertical retrace, updates the video memory buffer's palette, and handles polls the frame buffer window's message queue.

7.7.2 Window Initialization

The frame buffer window is initialized in the `initWindow()` function. This function is called whenever the window needs to change in some way, such as switching between full screen mode and windowed mode.

First, the function checks whether an old window exists, and if yes, it calls `deinitWindow()` which kills off the old window.

Then, normal window class registration is done. If full screen mode is desired, display settings change is requested from windows.

Next, the window is created, with slight differences between the full screen window and the normal windowed mode.

OpenGL is then initialized.

Finally, if full screen mode is used, the mouse is captured. If the new mode is a text mode, focus is changed back to the console window.

7.7.3 Frame Buffer Window Messages

The frame buffer window's messages are handled in `winproc()`.

The middle mouse button (or mouse wheel) is used to give the mouse focus to SolVBE, or to take it back. Additionally the middle mouse button is used to escape from the full screen mode.

Mouse events are analyzed to figure out how much the mouse has moved, and whether a mouse event is needed.

If the mouse has not been captured, the right mouse button is used to open the pop up menu. The menu can be used to change many of SolVBE's parameters at runtime.

If someone kills of the frame buffer window, we call `VDDTerminateVDM()` to kill off the VDM as well.

Any keyboard events are forwarded to the console window.

Window size events are tracked, and the OpenGL rendering target is resized accordingly.

If lose focus for any reason, we will release the mouse.

7.7.4 Video Memory

The video memory buffer is initialized with `initVideoMemory()`. The video memory buffer is always in the video mode's format, and its size is as wide as the mode's width, and approximately as high as the virtual video memory allows.

Whenever mode changes, we need to uninitialized the video memory buffer. This happens in `deinitFramebuffer()`.

7.7.5 Frame Buffer

The frame buffer is, like the video memory buffer, a GDI bitmap. Unlike the video memory buffer, the frame buffer is always a 32-bit buffer.

7.7.6 Rendering

When a screen update is required, the video memory buffer is blitted (with the current screen offset) to the frame buffer, causing the pixel format conversion to occur. Then the frame buffer is used to update the OpenGL texture, and finally the texture is used to draw two triangles on the screen. This happens in the `render()` function.